

Enciclopedia del lenguaje C++

2ª edición

Programación orientada a objetos

Elementos del lenguaje

Lógica de programación

Estructura de un programa



Más de 270 ejemplos resueltos

ra-ma.es

Fco. Javier Ceballos Sierra

Puede descargarse el CD-ROM con las URL para obtener el software de desarrollo y las aplicaciones contenidas en el libro



Ra-Ma®

Enciclopedia del lenguaje C++

2ª edición

Enciclopedia del lenguaje C++

2ª edición

Fco. Javier Ceballos Sierra

Profesor titular de la
Escuela Politécnica Superior
Universidad de Alcalá





Enciclopedia del lenguaje C++. 2ª edición.

© Fco. Javier Ceballos Sierra

© De la edición: RA-MA 2009

MARCAS COMERCIALES: Las marcas de los productos citados en el contenido de este libro (sean o no marcas registradas) pertenecen a sus respectivos propietarios. RA-MA no está asociada a ningún producto o fabricante mencionado en la obra, los datos y los ejemplos utilizados son ficticios salvo que se indique lo contrario.

RA-MA es una marca comercial registrada.

Se ha puesto el máximo empeño en ofrecer al lector una información completa y precisa. Sin embargo, RA-MA Editorial no asume ninguna responsabilidad derivada de su uso, ni tampoco por cualquier violación de patentes ni otros derechos de terceras partes que pudieran ocurrir. Esta publicación tiene por objeto proporcionar unos conocimientos precisos y acreditados sobre el tema tratado. Su venta no supone para el editor ninguna forma de asistencia legal, administrativa ni de ningún otro tipo. En caso de precisarse asesoría legal u otra forma de ayuda experta, deben buscarse los servicios de un profesional competente.

Reservados todos los derechos de publicación en cualquier idioma.

Según lo dispuesto en el Código Penal vigente ninguna parte de este libro puede ser reproducida, grabada en sistema de almacenamiento o transmitida en forma alguna ni por cualquier procedimiento, ya sea electrónico, mecánico, reprográfico, magnético o cualquier otro, sin autorización previa y por escrito de RA-MA; su contenido está protegido por la Ley vigente que establece penas de prisión y/o multas a quienes intencionadamente, reprodujeren o plagiaran, en todo o en parte, una obra literaria, artística o científica.

Editado por:

RA-MA Editorial

C/ Jarama, 3A, Polígono industrial Igarsa

28860 PARACUELLOS DEL JARAMA, Madrid

Teléfono: 91 658 42 80

Telefax: 91 662 81 39

Correo electrónico: editorial@ra-ma.com

Internet: ebooks.ra-ma.com, www.ra-ma.es y www.ra-ma.com

ISBN: 978-84-9964-357-1

Depósito Legal: M-15276-2009

Autoedición: Fco. Javier Ceballos

Filmación e impresión: Closas-Orcoyen S. L.

Impreso en España

Primera impresión: Abril 2009

La diferencia sólo se marca cuando te sales del camino ya marcado por otros.

*Dedico esta obra
a María del Carmen, mi esposa,
y a mis hijos Francisco y Javier.*

RESUMEN DEL CONTENIDO

PARTE 1. PROGRAMACIÓN BÁSICA	1
CAPÍTULO 1. INTRODUCCIÓN A C++	3
CAPÍTULO 2. ELEMENTOS DEL LENGUAJE C++	25
CAPÍTULO 3. ESTRUCTURA DE UN PROGRAMA	59
CAPÍTULO 4. ENTRADA Y SALIDA ESTÁNDAR	97
CAPÍTULO 5. SENTENCIAS DE CONTROL.....	127
CAPÍTULO 6. TIPOS ESTRUCTURADOS DE DATOS	165
CAPÍTULO 7. PUNTEROS, REFERENCIAS Y GESTIÓN DE LA MEMORIA .	225
CAPÍTULO 8. MÁS SOBRE FUNCIONES	281
PARTE 2. MECANISMOS DE ABSTRACCIÓN.....	333
CAPÍTULO 9. CLASES	335
CAPÍTULO 10. OPERADORES SOBRECARGADOS	421
CAPÍTULO 11. CLASES DERIVADAS	483
CAPÍTULO 12. PLANTILLAS	579
CAPÍTULO 13. EXCEPCIONES	623

CAPÍTULO 14. FLUJOS	661
PARTE 3. DISEÑO Y PROGRAMACIÓN	707
CAPÍTULO 15. ESTRUCTURAS DINÁMICAS	709
CAPÍTULO 16. ALGORITMOS	797
PARTE 4. APÉNDICES.....	813
A. NOVEDADES EN C++0x	815
B. LA BIBLIOTECA ESTÁNDAR DE C++.....	833
C. LA BIBLIOTECA DE C	837
D. ENTORNOS DE DESARROLLO	863
E. INSTALACIÓN DEL PAQUETE DE DESARROLLO	879
F. CÓDIGOS DE CARACTERES.....	883

CONTENIDO

PRÓLOGO	XXI
PARTE 1. PROGRAMACIÓN BÁSICA	1
CAPÍTULO 1. INTRODUCCIÓN A C++	3
¿POR QUÉ APRENDER C++?	4
REALIZACIÓN DE UN PROGRAMA EN C++	5
Cómo crear un programa.....	6
¿Qué hace este programa?.....	7
Guardar el programa escrito en el disco.....	8
Compilar y ejecutar el programa.....	8
Biblioteca estándar de C++	8
Depurar un programa	10
EJERCICIO	11
DECLARACIÓN DE UNA VARIABLE	12
ASIGNAR VALORES.....	15
AÑADIR COMENTARIOS	16
EXPRESIONES ARITMÉTICAS	18
EXPRESIONES CONDICIONALES.....	19
ESCRIBIR NUESTRAS PROPIAS FUNCIONES.....	22
EJERCICIOS PROPUESTOS.....	23
CAPÍTULO 2. ELEMENTOS DEL LENGUAJE C++	25
PRESENTACIÓN DE LA SINTAXIS DE C++.....	25
CARACTERES DE C++	26

Letras, dígitos y carácter de subrayado	26
Espacios en blanco	26
Caracteres especiales y signos de puntuación	27
Secuencias de escape.....	27
TIPOS DE DATOS	28
Tipos primitivos	28
Tipos derivados	29
Enumeraciones.....	30
Clases	32
SINÓNIMOS DE UN TIPO.....	33
LITERALES	33
Literales enteros	34
Literales reales	35
Literales de un solo carácter.....	35
Literales de cadenas de caracteres.....	36
IDENTIFICADORES	36
PALABRAS CLAVE.....	36
DECLARACIÓN DE CONSTANTES SIMBÓLICAS	37
¿Por qué utilizar constantes?.....	38
DECLARACIÓN DE UNA VARIABLE	38
Iniciación de una variable	39
OPERADORES.....	39
Operadores aritméticos.....	39
Operadores de relación.....	40
Operadores lógicos.....	41
Operadores unitarios	42
Operadores a nivel de bits.....	43
Operadores de asignación	43
Operador condicional	46
Otros operadores	47
Operador global y de resolución de ámbito (::)	47
Operador sizeof.....	48
Operador coma.....	48
Operador dirección-de	49
Operador de indirección.....	49
Operador referencia a.....	50
PRIORIDAD Y ORDEN DE EVALUACIÓN	52
CONVERSIÓN ENTRE TIPOS DE DATOS.....	53
EJERCICIOS PROPUESTOS.....	56

CAPÍTULO 3. ESTRUCTURA DE UN PROGRAMA	59
PARADIGMAS DE PROGRAMACIÓN	59
ESTRUCTURA DE UN PROGRAMA C++	60
Directrices para el preprocesador	63
Inclusión incondicional	64
Definición de un identificador	64
Inclusión condicional	65
Definiciones y declaraciones	65
Sentencia simple	66
Sentencia compuesta o bloque	66
Funciones	67
Declaración de una función	67
Definición de una función	70
Llamada a una función	72
Función main	72
PASANDO ARGUMENTOS A LAS FUNCIONES	73
PROGRAMA C++ FORMADO POR VARIOS MÓDULOS	78
ÁMBITO DE UN NOMBRE	81
Nombres globales y locales	81
CLASES DE ALMACENAMIENTO DE UNA VARIABLE	83
Calificación de variables globales	83
Calificación de variables locales	85
ESPACIOS DE NOMBRES	87
Directriz using	89
EJERCICIOS RESUELTOS	91
EJERCICIOS PROPUESTOS	92
CAPÍTULO 4. ENTRADA Y SALIDA ESTÁNDAR	97
ENTRADA Y SALIDA	98
Flujos de salida	99
Flujos de entrada	101
ESTADO DE UN FLUJO	104
DESCARTAR CARACTERES DEL FLUJO DE ENTRADA	106
ENTRADA/SALIDA CON FORMATO	107
ENTRADA DE CARACTERES	112
CARÁCTER FIN DE FICHERO	113
CARÁCTER \n	115
ENTRADA DE CADENAS DE CARACTERES	117
EJERCICIOS RESUELTOS	117
EJERCICIOS PROPUESTOS	122

CAPÍTULO 5. SENTENCIAS DE CONTROL..... 127

SENTENCIA if.....	127
ANIDAMIENTO DE SENTENCIAS if.....	129
ESTRUCTURA else if.....	132
SENTENCIA switch.....	134
SENTENCIA while.....	137
Bucles anidados.....	140
SENTENCIA do ... while.....	142
SENTENCIA for.....	145
SENTENCIA break.....	149
SENTENCIA continue.....	150
SENTENCIA goto.....	150
SENTENCIAS try ... catch.....	152
EJERCICIOS RESUELTOS.....	153
EJERCICIOS PROPUESTOS.....	157

CAPÍTULO 6. TIPOS ESTRUCTURADOS DE DATOS 165

INTRODUCCIÓN A LAS MATRICES.....	166
MATRICES NUMÉRICAS UNIDIMENSIONALES.....	167
Definir una matriz.....	167
Acceder a los elementos de una matriz.....	168
Iniciar una matriz.....	169
Trabajar con matrices unidimensionales.....	170
Tipo y tamaño de una matriz.....	172
Vector.....	172
Acceso a los elementos.....	173
Iteradores.....	174
Tamaño.....	175
Eliminar elementos.....	175
Buscar elementos.....	175
Insertar elementos.....	175
Ejemplo.....	176
Matrices asociativas.....	178
Map.....	180
CADENAS DE CARACTERES.....	183
Leer y escribir una cadena de caracteres.....	184
String.....	185
Constructores.....	185
Iteradores.....	186
Acceso a un carácter.....	186

Asignación	186
Conversiones a cadenas estilo C	187
Comparaciones.....	187
Inserción.....	188
Concatenación.....	189
Búsqueda.....	189
Reemplazar	189
Subcadenas.....	189
Tamaño	190
Operaciones de E/S.....	190
MATRICES MULTIDIMENSIONALES.....	191
Matrices numéricas multidimensionales	191
Matrices de cadenas de caracteres.....	197
Matrices de objetos string	198
SENTENCIA for_each.....	201
ESTRUCTURAS	201
Definir una estructura.....	202
Matrices de estructuras.....	206
UNIONES	208
EJERCICIOS RESUELTOS	210
EJERCICIOS PROPUESTOS.....	218

CAPÍTULO 7. PUNTEROS, REFERENCIAS Y GESTIÓN DE LA MEMORIA. 225

CREACIÓN DE PUNTEROS	225
Operadores	227
Importancia del tipo del objeto al que se apunta.....	228
OPERACIONES CON PUNTEROS	228
Operación de asignación	229
Operaciones aritméticas	229
Comparación de punteros.....	231
Punteros genéricos	231
Puntero nulo	232
Punteros y objetos constantes	232
REFERENCIAS	233
Paso de parámetros por referencia	233
PUNTEROS Y MATRICES	235
Punteros a cadenas de caracteres.....	239
MATRICES DE PUNTEROS.....	241
Punteros a punteros	243
Matriz de punteros a cadenas de caracteres	246
ASIGNACIÓN DINÁMICA DE MEMORIA	248

Operadores para asignación dinámica de memoria	249
new	249
delete	251
Reasignar un bloque de memoria	252
MATRICES DINÁMICAS	254
PUNTEROS A ESTRUCTURAS	258
PUNTEROS COMO PARÁMETROS EN FUNCIONES	260
DECLARACIONES COMPLEJAS	264
EJERCICIOS RESUELTOS	265
EJERCICIOS PROPUESTOS	272
CAPÍTULO 8. MÁS SOBRE FUNCIONES	281
PASAR UNA MATRIZ COMO ARGUMENTO A UNA FUNCIÓN	281
Matrices automáticas	282
Matrices dinámicas y contenedores	284
PASAR UN PUNTERO COMO ARGUMENTO A UNA FUNCIÓN	286
PASAR UNA ESTRUCTURA A UNA FUNCIÓN	290
DATOS RETORNADOS POR UNA FUNCIÓN	293
Retornar una copia de los datos	293
Retornar un puntero al bloque de datos	295
Retornar la dirección de una variable declarada static	297
Retornar una referencia	299
ARGUMENTOS EN LA LÍNEA DE ÓRDENES	301
REDIRECCIÓN DE LA ENTRADA Y DE LA SALIDA	303
FUNCIONES RECURSIVAS	305
PARÁMETROS POR OMISIÓN EN UNA FUNCIÓN	307
FUNCIONES EN LÍNEA	309
MACROS	310
FUNCIONES SOBRECARGADAS	311
Ambigüedades	313
OPERADORES SOBRECARGADOS	314
PUNTEROS A FUNCIONES	315
EJERCICIOS RESUELTOS	320
EJERCICIOS PROPUESTOS	325
PARTE 2. MECANISMOS DE ABSTRACCIÓN	333
CAPÍTULO 9. CLASES	335
DEFINICIÓN DE UNA CLASE	335

Atributos	337
Métodos de una clase	338
Control de acceso a los miembros de la clase	339
Acceso público	340
Acceso privado	341
Acceso protegido	341
Clases en ficheros de cabecera	341
IMPLEMENTACIÓN DE UNA CLASE	345
MÉTODOS SOBRECARGADOS	348
PARÁMETROS CON VALORES POR OMISIÓN	350
IMPLEMENTACIÓN DE UNA APLICACIÓN	351
EL PUNTERO IMPLÍCITO <code>this</code>	352
MÉTODOS Y OBJETOS CONSTANTES	354
INICIACIÓN DE UN OBJETO	356
Constructor	358
Asignación de objetos	362
Constructor copia	363
DESTRUCCIÓN DE OBJETOS	364
Destructor	365
PUNTEROS COMO ATRIBUTOS DE UNA CLASE	366
MIEMBROS STATIC DE UNA CLASE	375
Atributos <code>static</code>	375
Acceder a los atributos <code>static</code>	377
Métodos <code>static</code>	378
ATRIBUTOS QUE SON OBJETOS	380
CLASES INTERNAS	382
INTEGRIDAD DE LOS DATOS	384
DEVOLVER UN PUNTERO O UNA REFERENCIA	386
MATRICES DE OBJETOS	387
FUNCIONES AMIGAS DE UNA CLASE	397
PUNTEROS A LOS MIEMBROS DE UNA CLASE	400
EJERCICIOS RESUELTOS	404
EJERCICIOS PROPUESTOS	419
CAPÍTULO 10. OPERADORES SOBRECARGADOS	421
SOBRECARGAR UN OPERADOR	421
UNA CLASE PARA NÚMEROS RACIONALES	428
SOBRECARGA DE OPERADORES BINARIOS	430
Sobrecarga de operadores de asignación	430
Sobrecarga de operadores aritméticos	432
Aritmética mixta	434

Sobrecarga de operadores de relación	436
Métodos adicionales	436
Sobrecarga del operador de inserción	437
Sobrecarga del operador de extracción	440
SOBRECARGA DE OPERADORES UNARIOS	442
Incremento y decremento	442
Operadores unarios/binarios	444
CONVERSIÓN DE TIPOS DEFINIDOS POR EL USUARIO	444
Conversión mediante constructores	446
Operadores de conversión	447
Ambigüedades	451
ASIGNACIÓN	451
INDEXACIÓN	453
LLAMADA A FUNCIÓN	454
DESREFERENCIA	456
SOBRECARGA DE LOS OPERADORES new y delete	458
Sobrecarga del operador new	458
Sobrecarga del operador delete	461
EJERCICIOS RESUELTOS	463
EJERCICIOS PROPUESTOS	480

CAPÍTULO 11. CLASES DERIVADAS..... 483

CLASES DERIVADAS Y HERENCIA	484
DEFINIR UNA CLASE DERIVADA	488
Control de acceso a la clase base	489
Control de acceso a los miembros de las clases	490
Qué miembros hereda una clase derivada	491
ATRIBUTOS CON EL MISMO NOMBRE	496
REDEFINIR MÉTODOS DE LA CLASE BASE	498
CONSTRUCTORES DE CLASES DERIVADAS	500
COPIA DE OBJETOS	503
DESTRUCTORES DE CLASES DERIVADAS	506
JERARQUÍA DE CLASES	506
FUNCIONES AMIGAS	514
PUNTEROS Y REFERENCIAS	516
Conversiones implícitas	517
Restricciones	519
Conversiones explícitas	520
MÉTODOS VIRTUALES	522
Cómo son implementados los métodos virtuales	526
Constructores virtuales	528

Destructores virtuales.....	530
INFORMACIÓN DE TIPOS DURANTE LA EJECUCIÓN	532
Operador <code>dynamic_cast</code>	532
Operador <code>typeid</code>	535
POLIMORFISMO.....	535
CLASES ABSTRACTAS	550
HERENCIA MÚLTIPLE	552
Clases base virtuales	556
Redefinición de métodos de bases virtuales.....	560
Conversiones entre clases	562
EJERCICIOS RESUELTOS	563
EJERCICIOS PROPUESTOS.....	575
CAPÍTULO 12. PLANTILLAS	579
DEFINICIÓN DE UNA PLANTILLA	580
FUNCIONES GENÉRICAS	582
Especialización de plantillas de función	586
Sobrecarga de plantillas de función	588
ORGANIZACIÓN DEL CÓDIGO DE LAS PLANTILLAS	590
Fichero único.....	590
Fichero de declaraciones y fichero de definiciones	591
Fichero único combinación de otros	593
CLASES GENÉRICAS.....	594
Declaración previa de una clase genérica	599
Especialización de plantillas de clase.....	599
Derivación de plantillas	605
Otras características de las plantillas.....	608
EJERCICIOS RESUELTOS	611
EJERCICIOS PROPUESTOS.....	620
CAPÍTULO 13. EXCEPCIONES	623
EXCEPCIONES DE C++	625
MANEJAR EXCEPCIONES.....	628
Lanzar una excepción.....	629
Capturar una excepción.....	629
Excepciones derivadas	631
Capturar cualquier excepción.....	632
Relanzar una excepción.....	633
CREAR EXCEPCIONES	633
Especificación de excepciones	634

Excepciones no esperadas	635
FLUJO DE EJECUCIÓN	637
CUÁNDO UTILIZAR EXCEPCIONES Y CUÁNDO NO	642
ADQUISICIÓN DE RECURSOS	643
Punteros inteligentes	649
EJERCICIOS RESUELTOS	653
EJERCICIOS PROPUESTOS	659
CAPÍTULO 14. FLUJOS.....	661
VISIÓN GENERAL DE LOS FLUJOS DE E/S.....	663
BÚFERES	664
VISIÓN GENERAL DE UN FICHERO.....	666
DESCRIPCIÓN DE LOS BÚFERES Y FLUJOS	670
Clase streambuf.....	670
Clase filebuf	671
Clase ostream	673
Clase istream	675
Clase istream	678
Clase ostream.....	679
Clase ifstream.....	681
Clasefstream.....	683
E/S UTILIZANDO REGISTROS	685
ESCRIBIR DATOS EN LA IMPRESORA	687
ABRIENDO FICHEROS PARA ACCESO SECUENCIAL.....	687
Un ejemplo de acceso secuencial	688
ACCESO ALEATORIO A FICHEROS EN EL DISCO	698
EJERCICIOS PROPUESTOS.....	703
PARTE 3. DISEÑO Y PROGRAMACIÓN	707
CAPÍTULO 15. ESTRUCTURAS DINÁMICAS.....	709
LISTAS LINEALES	710
Listas lineales simplemente enlazadas	710
Operaciones básicas	713
Inserción de un elemento al comienzo de la lista.....	714
Buscar en una lista un elemento con un valor x.....	715
Inserción de un elemento en general.....	716
Borrar un elemento de la lista	717
Recorrer una lista	718

Borrar todos los elementos de una lista	718
UNA CLASE PARA LISTAS LINEALES	719
Clase genérica para listas lineales	722
Consistencia de la aplicación	731
LISTAS CIRCULARES	733
Clase CListaCircularSE<T>.....	734
PILAS.....	739
COLAS.....	741
EJEMPLO	743
LISTA DOBLEMENTE ENLAZADA.....	746
Lista circular doblemente enlazada.....	746
Clase CListaCircularDE<T>.....	747
Ejemplo.....	754
ÁRBOLES.....	756
Árboles binarios	757
Formas de recorrer un árbol binario.....	759
ÁRBOLES BINARIOS DE BÚSQUEDA.....	761
Clase CARbolBinB<T>	762
Buscar un nodo en el árbol.....	766
Insertar un nodo en el árbol.....	767
Borrar un nodo del árbol.....	768
Utilización de la clase CARbolBinB<T>.....	771
ÁRBOLES BINARIOS PERFECTAMENTE EQUILIBRADOS.....	774
Clase CARbolBinE<T>.....	775
Utilización de la clase CARbolBinE<T>.....	783
CLASES RELACIONADAS DE LA BIBLIOTECA C++.....	786
Plantilla list	786
EJERCICIOS PROPUESTOS.....	789
CAPÍTULO 16. ALGORITMOS.....	797
ORDENACIÓN DE DATOS.....	797
Método de la burbuja	798
Método de inserción.....	801
Método quicksort	802
Comparación de los métodos expuestos.....	804
BÚSQUEDA DE DATOS	805
Búsqueda secuencial	805
Búsqueda binaria.....	805
Búsqueda de cadenas	806
CLASES RELACIONADAS DE LA BIBLIOTECA C++.....	810
Modo de empleo de los algoritmos	811

EJERCICIOS PROPUESTOS.....	812
PARTE 4. APÉNDICES.....	813
NOVEDADES DE C++0x.....	815
LA BIBLIOTECA ESTÁNDAR DE C++	833
LA BIBLIOTECA DE C.....	837
ENTORNOS DE DESARROLLO	863
INSTALACIÓN DEL PAQUETE DE DESARROLLO	879
CÓDIGOS DE CARACTERES	883
ÍNDICE	889

PRÓLOGO

Un programa tradicional se compone de procedimientos y de datos. Un programa orientado a objetos consiste solamente en objetos, entendiendo por objeto una entidad que tiene unos atributos particulares, los datos, y unas formas de operar sobre ellos, los métodos o procedimientos.

La programación orientada a objetos es una de las técnicas más modernas que trata de disminuir el coste del software, aumentando la eficiencia en la programación y reduciendo el tiempo necesario para el desarrollo de una aplicación. Con la programación orientada a objetos, los programas tienen menos líneas de código, menos sentencias de bifurcación, y módulos que son más comprensibles porque reflejan de una forma clara la relación existente entre cada concepto a desarrollar y cada objeto que interviene en dicho desarrollo. Donde la programación orientada a objetos toma verdadera ventaja es en la compartición y reutilización del código.

Sin embargo, no debe pensarse que la programación orientada a objetos resuelve todos los problemas de una forma sencilla y rápida. Para conseguir buenos resultados, es preciso dedicar un tiempo significativamente superior al análisis y al diseño. No obstante, éste no es un tiempo perdido, ya que simplificará enormemente la realización de aplicaciones futuras.

Según lo expuesto, las ventajas de la programación orientada a objetos son sustanciales. Pero también presenta inconvenientes; por ejemplo, la ejecución de un programa no gana en velocidad y obliga al usuario a aprenderse una amplia biblioteca de clases antes de empezar a manipular un lenguaje orientado a objetos.

Existen varios lenguajes que permiten escribir un programa orientado a objetos y entre ellos se encuentra C++. Se trata de un lenguaje de programación basa-

do en el lenguaje C, estandarizado (ISO/IEC – *International Organization for Standardization/International Electrotechnical Commission*) y ampliamente difundido. Gracias a esta estandarización y a la biblioteca estándar, C++ se ha convertido en un lenguaje potente, eficiente y seguro, características que han hecho de él un lenguaje universal de propósito general ampliamente utilizado, tanto en el ámbito profesional como en el educativo, y competitivo frente a otros lenguajes como C# de Microsoft o Java de Sun Microsystems. Evidentemente, algunas nuevas características que se han incorporado a C# o a Java no están soportadas en la actualidad, como es el caso de la recolección de basura; no obstante, existen excelentes recolectores de basura de C++, tanto comerciales como gratuitos, que resuelven este problema. Otro futuro desarrollo previsto es la ampliación de la biblioteca estándar para desarrollar aplicaciones con interfaz gráfica de usuario.

¿Por qué C++? Porque posee características superiores a otros lenguajes. Las más importantes son:

- *Programación orientada a objetos.* Esta característica permite al programador diseñar aplicaciones pensando más bien en la comunicación entre objetos que en una secuencia estructurada de código. Además, permite la reutilización del código de una forma más lógica y productiva.
- *Portabilidad.* Prácticamente se puede compilar el mismo código C++ en la casi totalidad de ordenadores y sistemas operativos sin apenas hacer cambios. Por eso C++ es uno de los lenguajes más utilizados y portados a diferentes plataformas.
- *Brevedad.* El código escrito en C++ es muy corto en comparación con otros lenguajes, debido a la facilidad con la que se pueden anidar expresiones y a la gran cantidad de operadores.
- *Programación modular.* El cuerpo de una aplicación en C++ puede construirse a partir de varios ficheros fuente que serán compilados separadamente para después ser enlazados todos juntos. Esto supone un ahorro de tiempo durante el diseño, ya que cada vez que se realice una modificación en uno de ellos no es necesario recompilar la aplicación completa, sino sólo el fichero que se modificó.
- *Compatibilidad con C.* Cualquier código escrito en C puede fácilmente ser incluido en un programa C++ sin apenas cambios.
- *Velocidad.* El código resultante de una compilación en C++ es muy eficiente debido a su dualidad como lenguaje de alto y bajo nivel y al reducido tamaño del lenguaje mismo.

El libro, en su totalidad, está dedicado al aprendizaje del lenguaje C++, de la programación orientada a objetos y al desarrollo de aplicaciones. Esta materia puede agruparse en los siguientes apartados:

- Programación básica
- Mecanismos de abstracción
- Diseño y programación

La primera parte está pensada para que en poco tiempo pueda convertirse en programador de aplicaciones C++. Y para esto, ¿qué necesita? Pues simplemente leer ordenadamente los capítulos del libro, resolviendo cada uno de los ejemplos que en ellos se detallan. La segunda parte abarca en profundidad la programación orientada a objetos.

En la primera parte el autor ha tratado de desarrollar aplicaciones sencillas, para introducirle más bien en el lenguaje y en el manejo de la biblioteca de clases de C++, que en el diseño de clases de objetos. No obstante, después de su estudio sí debe haber quedado claro que un programa orientado a objetos sólo se compone de objetos. Es hora pues de entrar con detalle en la programación orientada a objetos, segunda parte, la cual tiene un elemento básico: la *clase*.

Pero si el autor finalizara el libro con las dos partes anteriores, privaría al lector de saber que C++ aún proporciona mucho más. Por eso la tercera parte continúa con otros capítulos dedicados a la implementación de estructuras dinámicas, al diseño de algoritmos y a la programación con hilos.

Todo ello se ha documentado con abundantes problemas resueltos. Cuando complete todas las partes, todavía no sabrá todo lo que es posible hacer con C++, pero sí habrá dado un paso importante.

Esta obra fue escrita utilizando un compilador GCC para Win32 (un compilador C++ de la colección de compiladores GNU) que se adjunta en el CD-ROM que acompaña al libro. Se trata de un compilador de libre distribución que cumple la norma ISO/IEC, del cual existen versiones para prácticamente todos los sistemas operativos. Por lo tanto, los ejemplos de este libro están escritos en C++ puro, tal y como se define en el estándar C++, lo que garantizará que se ejecuten en cualquier implementación que se ajuste a este estándar, que en breve serán la totalidad de las existentes. Por ejemplo, el autor probó la casi totalidad de los desarrollos bajo el paquete Microsoft Visual Studio .NET, y también sobre la plataforma Linux, para conseguir un código lo más portable posible.

Sobre los ejemplos del libro

La imagen del CD de este libro, con las aplicaciones desarrolladas y el software para reproducirlas, puede descargarla desde:

<https://www.tecno-libro.es/ficheros/descargas/9788499643571.zip>

La descarga consiste en un fichero ZIP con una contraseña ddd-dd-dddd-ddd-d que se corresponde con el ISBN de este libro (teclea los dígitos y los guiones).

Agradecimientos

En la preparación de este libro quiero, en especial, expresar mi agradecimiento a **Manuel Peinado Gallego**, profesor de la Universidad de Alcalá con una amplia experiencia en desarrollos con C++, porque revisó la primera edición de este libro; y a **Óscar García Población, Elena Campo Montalvo, Sebastián Sánchez Prieto, Inmaculada Rodríguez Santiago** y **M^a Dolores Rodríguez Moreno**, que basándose en su experiencia docente me hicieron diversas sugerencias sobre los temas tratados. Todos ellos son profesores de Universidad, con una amplia experiencia sobre la materia que trata el libro.

Finalmente, no quiero olvidarme del resto de mis compañeros, aunque no cite sus nombres, porque todos ellos, de forma directa o indirecta, me ayudaron con la crítica constructiva que hicieron sobre otras publicaciones anteriores a ésta, y tampoco de mis alumnos, que con su interés por aprender me hacen reflexionar sobre la forma más adecuada de transmitir estos conocimientos; a todos ellos les estoy francamente agradecido.

Francisco Javier Ceballos Sierra

<http://www.fjceballos.es/>

P A R T E

1

Programación básica

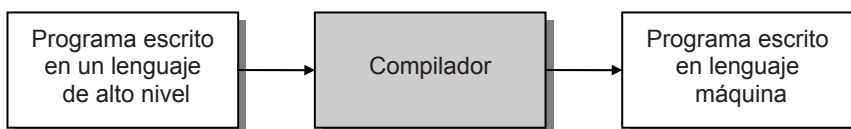
- Introducción a C++
- Elementos del lenguaje
- Estructura de un programa
- Entrada y salida estándar
- Sentencias de control
- Tipos estructurados de datos
- Punteros, referencias y gestión de la memoria
- Más sobre funciones

INTRODUCCIÓN A C++

Un *programa* no es nada más que una serie de instrucciones dadas al ordenador en un lenguaje entendido por él, para decirle exactamente lo que queremos que haga. Si el ordenador no entiende alguna instrucción, lo comunicará generalmente mediante mensajes visualizados en la pantalla.

Un programa tiene que escribirse en un lenguaje entendible por el ordenador. Desde el punto de vista físico, un ordenador es una máquina electrónica. Los elementos físicos (memoria, unidad central de proceso, etc.) de que dispone el ordenador para representar las instrucciones y los datos son de tipo binario; esto es, cada elemento puede diferenciar dos estados (dos niveles de voltaje). Cada estado se denomina genéricamente *bit* y se simboliza por *0* ó *1*. Por lo tanto, para representar y manipular información numérica, alfabética y alfanumérica se emplean cadenas de *bits*. Según esto, se denomina *byte* a la cantidad de información empleada por un ordenador para representar un carácter; generalmente un *byte* es una cadena de ocho *bits*. Esto hace pensar que escribir un programa utilizando ceros y unos (lenguaje máquina) llevaría mucho tiempo y con muchas posibilidades de cometer errores. Por este motivo, se desarrollaron los lenguajes de programación.

Para traducir un programa escrito en un determinado lenguaje de programación a lenguaje máquina (código binario), se utiliza un programa llamado *compilador* que ejecutamos mediante el propio ordenador. Este programa tomará como datos nuestro programa escrito en un lenguaje de alto nivel, por ejemplo en C++, y dará como resultado el mismo programa pero escrito en lenguaje máquina, lenguaje que entiende el ordenador.



¿POR QUÉ APRENDER C++?

Una de las ventajas de C++ es su independencia de la plataforma en lo que a código fuente se refiere. Otra característica importante de C++ es que es un lenguaje que soporta diversos estilos de programación (por ejemplo, la programación genérica y la programación orientada a objetos –POO– de la cual empezaremos a hablar en este mismo capítulo). Todos los estilos se basan en una verificación fuerte de tipos y permiten alcanzar un alto nivel de abstracción.

C++ está organizado de tal forma que el aprendizaje del mismo puede hacerse gradualmente obteniendo beneficios prácticos a largo de este camino. Esto es importante, porque podemos ir produciendo proporcionalmente a lo aprendido.

C++ está fundamentado en C lo que garantiza que los millones de líneas de código C existentes puedan beneficiarse de C++ sin necesidad de reescribirlas. Evidentemente, no es necesario aprender C para aprender C++, lo comprobará con este libro. No obstante, si conoce C, podrá comprobar que C++ es más seguro, más expresivo y reduce la necesidad de tener que centrarse en ideas de bajo nivel.

C++ se utiliza ampliamente en docencia e investigación porque es claro, realista y eficiente. También es lo suficientemente flexible como para realizar los proyectos más exigentes. Y también es lo suficientemente comercial como para ser incorporado en el desarrollo empresarial.

Existen varias implementaciones de C++, de distribución gratuita; por ejemplo, GCC. Las siglas GCC significan *GNU Compiler Collection* (colección de compiladores GNU; antes significaban *GNU C Compiler*: compilador C GNU). Como su nombre indica es una colección de compiladores y admite diversos lenguajes: C, C++, Objective C, Fortran, Java, etc. Existen versiones para prácticamente todos los sistemas operativos y pueden conseguirse en gcc.gnu.org.

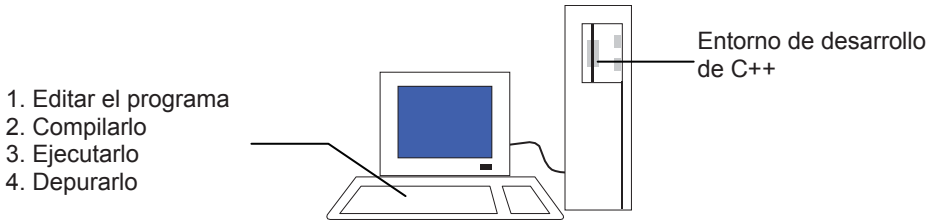
GNU (se trata de un acrónimo recursivo para “Gnu No es Unix”) es un proyecto que comenzó en 1984 para desarrollar un sistema operativo tipo Unix que fuera libre; lo que hoy en día conocemos como Linux es un sistema operativo GNU, aunque sería más preciso llamarlo GNU/Linux. Pues bien, dentro de este proyecto se desarrolló GCC, cuyo compilador C++ se ajusta al estándar ISO/IEC.

La mayor parte del software libre está protegido por la licencia pública GNU denominada GPL (*GNU Public License*).

REALIZACIÓN DE UN PROGRAMA EN C++

En este apartado se van a exponer los pasos a seguir en la realización de un programa, por medio de un ejemplo.

La siguiente figura muestra de forma esquemática lo que un usuario de C++ necesita y debe hacer para desarrollar un programa.

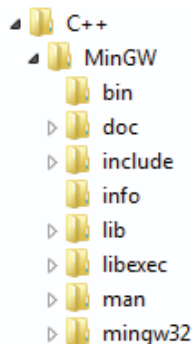


Evidentemente, para poder escribir programas se necesita un entorno de desarrollo C++. En Internet puede encontrar varios con licencia pública GNU que se distribuyen gratuitamente. Por ejemplo, el compilador C++ de GCC se puede obtener en la dirección de Internet:

<http://gcc.gnu.org>

Así mismo, el CD-ROM que acompaña al libro incluye *MinGW*, una versión nativa de Win32 de GCC (para Windows 2000/XP/Vista). Linux también incluye una implementación GCC.

Para instalar la implementación *MinGW* de GCC incluida en el CD en una plataforma Windows, descargue el fichero *MinGW-x.x.x.exe*, o bien utilice la versión suministrada en el CD del libro y ejecútelo. Después, siga los pasos especificados por el programa de instalación. Puede ver más detalles sobre la instalación en los apéndices del libro. Una vez finalizada la instalación, suponiendo que la realizó en la carpeta C++, se puede observar el siguiente contenido:



- La carpeta *bin* contiene las herramientas de desarrollo. Esto es, los programas para compilar (*gcc* permite compilar un programa C, *g++* permite compilar un programa C++, etc.), depurar (*gdb*), y otras utilidades.
- La carpeta *include* contiene los ficheros de cabecera de C.
- La carpeta *doc* contiene información de ayuda acerca de la implementación *MinGW*.
- La carpeta *lib* contiene bibliotecas de clases, de funciones y ficheros de soporte requeridos por las herramientas de desarrollo.
- La carpeta *mingw32* contiene otras carpetas *bin* y *lib* con otros ficheros adicionales.

Sólo falta un editor de código fuente C++. Es suficiente con un editor de texto sin formato. No obstante, todo el trabajo de edición, compilación, ejecución y depuración se hará mucho más fácil si se utiliza un entorno de desarrollo con interfaz gráfica de usuario que integre las herramientas mencionadas, en lugar de tener que utilizar la interfaz de línea de órdenes del entorno de desarrollo C++ instalado, como veremos a continuación.

Entornos de desarrollo integrados para C++ hay varios: *Microsoft Visual Studio*, *CodeBlocks*, *Eclipse*, *NetBeans*, etc. Concretamente en el CD se proporciona *CodeBlocks*: un entorno integrado, con licencia pública GNU, y que se ajusta a las necesidades de las aplicaciones que serán expuestas en este libro. Para más detalle véase en los apéndices *Instalación del paquete de desarrollo*.

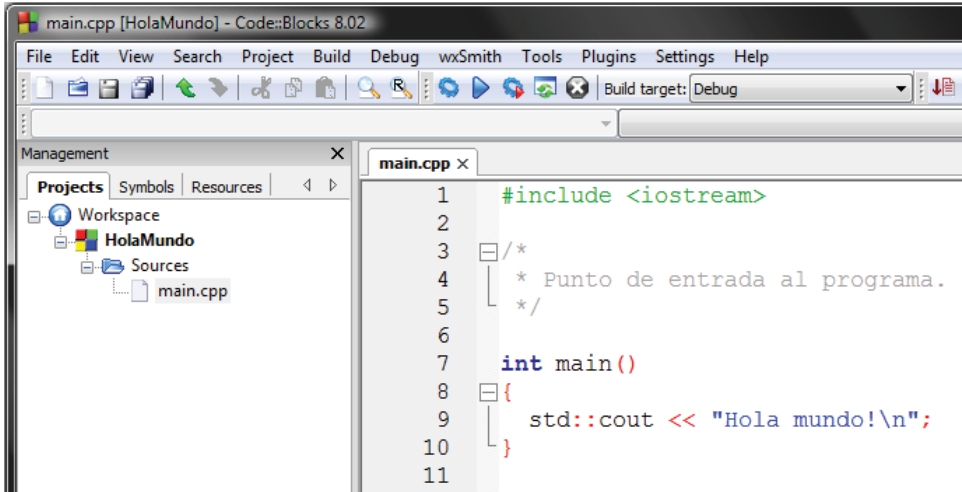
Cómo crear un programa

Empecemos con la creación de un programa sencillo: el clásico ejemplo de mostrar un mensaje de saludo.

Este sencillo programa lo realizaremos utilizando el entorno de desarrollo integrado *CodeBlocks*. No obstante, una vez editado el fichero fuente C++, podría también compilarlo y ejecutarlo desde la línea de órdenes, aspecto que puede ver con detalle en el apéndice *Entornos de desarrollo* del libro.

Empecemos por editar el fichero fuente C++ correspondiente al programa. Primeramente pondremos en marcha el EDI, en nuestro caso *CodeBlocks*. Después, creamos un nuevo proyecto, por ejemplo *HolaMundo*, con un fichero *main.cpp* (el nombre del fichero puede ser cualquiera, pero la extensión debe ser

.cpp) y lo editamos como muestra la figura siguiente (para más detalles, véase el apéndice *Entornos de desarrollo* del libro):



¿Qué hace este programa?

Comentamos brevemente cada línea de este programa. No apurarse si algunos de los términos no quedan muy claros ya que todos ellos se verán con detalle en capítulos posteriores.

La primera línea incluye el fichero de cabecera *iostream* que contiene las declaraciones necesarias para que se puedan ejecutar las sentencias de entrada o salida (E/S) que aparecen en el programa; en nuestro caso para **cout**. Esto significa que, como regla general, antes de invocar a algún elemento de la biblioteca de C++ este tiene que estar declarado. Las palabras reservadas de C++ que empiezan con el símbolo # reciben el nombre de *directrices* del compilador y son procesadas por el *preprocesador* de C++ cuando se invoca al compilador, pero antes de iniciarse la compilación.

Las siguientes líneas encerradas entre */** y **/* son simplemente un comentario. Los comentarios no son tenidos en cuenta por el compilador, pero ayudan a entender un programa cuando se lee.

A continuación se escribe la función principal **main**. Todo programa escrito en C++ tiene una función **main**. Observe que una función se distingue por el modificador () que aparece después de su nombre y que el cuerpo de la misma empieza con el carácter { y finaliza con el carácter }. Las llaves, {}, delimitan el bloque de código que define las acciones que tiene que ejecutar dicha función.

Cuando se ejecuta un programa, C++ espera que haya una función **main**. Esta función define el punto de entrada y de salida normal del programa.

El objeto **std::cout** de la biblioteca C++ escribe en la salida estándar (la pantalla) las expresiones que aparecen a continuación de los operadores << (operador de inserción). En nuestro caso, escribe la cadena de caracteres especificada entre comillas, *Hola mundo!*, y un retorno de carro (CR) más un avance a la línea siguiente (LF) que es lo que indica la constante `\n`. Observe que la sentencia completa finaliza con punto y coma.

Guardar el programa escrito en el disco

El programa editado está ahora en la memoria. Para que este trabajo pueda tener continuidad, el programa escrito se debe grabar en el disco utilizando la orden correspondiente del editor. Muy importante: el nombre del programa fuente debe añadir la extensión *cpp* (*c plus plus: c más más*), o bien *cxx*.

Compilar y ejecutar el programa

El siguiente paso es *compilar* el programa; esto es, traducir el programa fuente a lenguaje máquina para posteriormente enlazarlo con los elementos necesarios de la biblioteca de C++, proceso que generalmente se realiza automáticamente, y obtener así un programa ejecutable. Por ejemplo, en el entorno de desarrollo de C++ que hemos instalado, ejecutaremos la orden *Build* del menú *Build*.

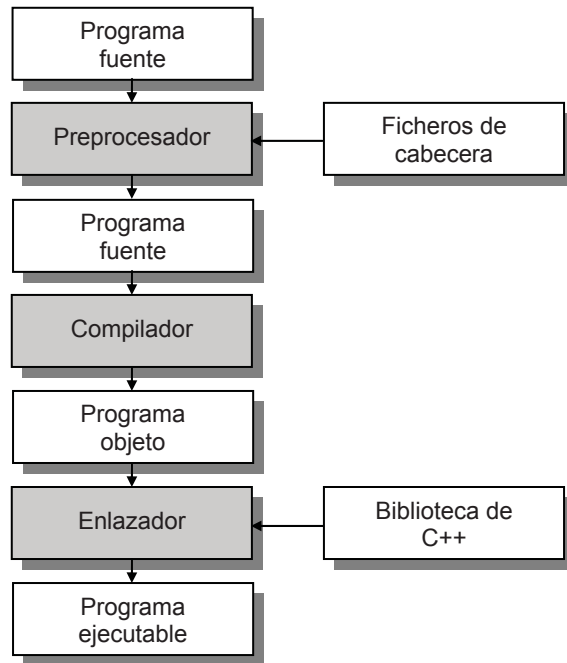
Al compilar un programa se pueden presentar *errores de compilación*, debidos a que el programa escrito no se adapta a la sintaxis y reglas del compilador. Estos errores se irán corrigiendo hasta obtener una compilación sin errores.

Para ejecutar el fichero resultante, en el entorno de desarrollo de C++ que estamos utilizando, ejecutaremos la orden *Run* del menú *Build*.

Biblioteca estándar de C++

C++ carece de instrucciones de E/S, de instrucciones para manejo de cadenas de caracteres, etc., con lo que este trabajo queda para la biblioteca de funciones genéricas y clases provista con el compilador. Una función es un conjunto de instrucciones que realizan una tarea específica. Una clase es un tipo de objetos. Una biblioteca es un fichero separado en el disco (generalmente con extensión *.lib*, típica de Windows, o con extensión *.a*, típica de LINUX) que contiene las clases y funciones genéricas que realizan las tareas más comunes, para que nosotros no tengamos que escribirlas. Como ejemplo, hemos visto anteriormente el objeto **std::cout**. Si este objeto no existiera, sería labor nuestra el escribir el código nece-

sario para visualizar los resultados sobre la pantalla. Toda la biblioteca estándar de C++ está definida en un único espacio de nombres llamado **std** (estándar). Por eso se escribió **std::cout** en vez de **cout**; como explicaremos en un capítulo posterior, los espacios de nombres son un mecanismo para evitar colisiones entre los nombres dados a los elementos que intervienen en un programa. Estos elementos de la biblioteca de C++ utilizados en nuestro programa serán incluidos en el fichero ejecutable por el *enlazador* (*linker*) sólo si no se produjeron errores de compilación. En la figura siguiente se muestran los pasos seguidos, en este caso por el EDI, para obtener un programa ejecutable:



Otra forma de utilizar elementos de la biblioteca estándar de C++ es indicando explícitamente mediante la directriz **using** el espacio de nombres al que pertenecen, en lugar de poner como prefijo para cada uno de ellos dicho espacio. El ejemplo siguiente muestra cómo sería el programa anterior utilizando esta técnica:

```

#include <iostream>
using namespace std;
/*
 * Punto de entrada al programa.
 */
int main()
{
    cout << "iiiHola mundo!!!\n";
}
  
```

Cuando se crea un fichero ejecutable, primero se utiliza el compilador C++ para compilar el programa fuente, el cual puede estar formado por uno o más ficheros *.cpp*, dando lugar a uno o más ficheros intermedios conocidos como ficheros objeto (con extensión *.obj*, típica de Windows, o *.o*, típica de LINUX) ya que cada *cpp* se compila por separado. A continuación se utiliza el programa *enlazador* (*linker*) para unir en un único fichero ejecutable el fichero o ficheros objeto y los elementos de la biblioteca estándar de C++ que el programa utilice.

Cuando se hace referencia a un elemento externo (no definido en el programa, sino en una biblioteca externa), como sucede con el objeto **cout**, el enlazador hará una de estas dos cosas: buscará la definición del elemento, primero en los módulos objeto del programa (ficheros *.obj* o *.o*) y después en la biblioteca, y si lo encuentra, la referencia estará resuelta, y si no, añadirá el identificador del mismo a la lista de “referencias no resueltas” para comunicárselo al usuario.

Según lo expuesto, cada vez que se realiza el proceso de *compilación* y *enlace* del programa actual, C++ genera automáticamente sobre el disco un fichero ejecutable. Este fichero puede ser ejecutado directamente desde el sistema operativo sin el soporte de C++, escribiendo el nombre del fichero a continuación del símbolo del sistema (*prompt* del sistema) y pulsando la tecla *Entrar*.

Al ejecutar el programa, pueden ocurrir *errores durante la ejecución*. Por ejemplo, puede darse una división por 0. Estos errores solamente pueden ser detectados cuando se ejecuta el programa y serán notificados con el correspondiente mensaje de error.

Hay *otro tipo de errores* que no dan lugar a mensaje alguno. Por ejemplo, un programa que no termine nunca de ejecutarse, debido a que presenta un lazo donde no se llega a dar la condición de terminación. Para detener la ejecución se tienen que pulsar las teclas *Ctrl+C* en Windows o *Ctrl+D* en Linux.

Depurar un programa

Una vez ejecutado el programa, la solución puede ser incorrecta. Este caso exige un análisis minucioso de cómo se comporta el programa a lo largo de su ejecución; esto es, hay que entrar en la fase de *depuración* del programa.

La forma más sencilla y eficaz para realizar este proceso es utilizar un programa *depurador*. En el apéndice *Entornos de desarrollo* se explica cómo utilizar un depurador desde un EDI.

EJERCICIO

Para practicar con un programa más, escriba el siguiente ejemplo y pruebe los resultados. Este ejemplo visualiza como resultado la suma, la resta, la multiplicación y la división de dos cantidades enteras.

Abra el entorno de desarrollo integrado (EDI), cree un proyecto, por ejemplo *Aritmetica*, y edite el programa ejemplo que se muestra a continuación. Recuerde, el nombre del fichero fuente debe tener extensión *.cpp*, por ejemplo *main.cpp*.

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

/*
 * Operaciones aritméticas
 */

int main()
{
    int dato1, dato2, resultado;

    dato1 = 20;
    dato2 = 10;

    // Suma
    resultado = dato1 + dato2;
    cout << dato1 << " + " << dato2 << " = " << resultado << endl;

    // Resta
    resultado = dato1 - dato2;
    cout << dato1 << " - " << dato2 << " = " << resultado << endl;

    // Producto
    resultado = dato1 * dato2;
    cout << dato1 << " * " << dato2 << " = " << resultado << endl;

    // Cociente
    resultado = dato1 / dato2;
    cout << dato1 << " / " << dato2 << " = " << resultado << endl;
}
```

La constante **endl** (final de línea) hace la misma acción que **\n** y además, vacía el *buffer* de la salida estándar. Una vez editado el programa, guárdelo en el disco con el nombre *Aritmetica.cpp*.

¿Qué hace este programa?

Fijándonos en la función principal, **main**, vemos que se han declarado tres variables enteras (de tipo **int**): *dato1*, *dato2* y *resultado*.

```
int dato1, dato2, resultado;
```

Las líneas que comienzan con `//` son comentarios estilo C++.

El siguiente paso asigna el valor 20 a la variable *dato1* y el valor 10 a la variable *dato2*.

```
dato1 = 20;  
dato2 = 10;
```

A continuación se realiza la suma de esos valores y se escriben los datos y el resultado.

```
resultado = dato1 + dato2;  
cout << dato1 << " + " << dato2 << " = " << resultado << endl;
```

El objeto **cout** escribe un resultado de la forma:

```
20 + 10 = 30
```

Observe que la expresión resultante está formada por seis elementos: *dato1*, `" + "`, *dato2*, `" = "`, *resultado* y **endl**. Unos elementos son numéricos y otros son constantes de caracteres. Para mostrar cada uno de los seis elementos se ha empleado el operador de inserción `<<`.

Un proceso similar se sigue para calcular la diferencia, el producto y el cociente.

Para finalizar, compile, ejecute el programa y observe los resultados.

DECLARACIÓN DE UNA VARIABLE

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo, valor que puede ser modificado a lo largo de la ejecución del bloque donde la variable es accesible, tantas veces como se necesite. La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo: el tipo del valor que va a almacenar. Por ejemplo, el siguiente código declara cuatro variables: *a* de tipo **double**, *b* de tipo **float**, y *c* y *r* de tipo **int**:

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

int main()
{
    double a;
    float b;
    int c, r;

    // ...
}
```

Por definición, una variable declarada dentro de un bloque, entendiendo por bloque el código encerrado entre los caracteres ‘{’ y ‘}’, es accesible sólo dentro de ese bloque. Más adelante, cuando tratemos con objetos matizaremos el concepto de accesibilidad.

Según lo expuesto, las variables *a*, *b*, *c* y *r* son accesibles sólo desde la función **main**. En este caso se dice que dichas variables son *locales* al bloque donde han sido declaradas. Una variable local se crea cuando se ejecuta el bloque donde se declara y se destruye cuando finaliza la ejecución de dicho bloque.

Las variables locales no son iniciadas por el compilador C++. Por lo tanto, es aconsejable iniciarlas para evitar resultados inesperados.

```
int main()
{
    double a = 0;
    float b = 0;
    int c = 0, r = 0;

    // ...

    cout << a << ", " << b << ", " << c << ", " << r << endl;
}
```

Cuando elija el identificador para declarar una variable, tenga presente que el compilador C++ trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo las variables *dato1* y *Dato1* son diferentes.

Respecto al tipo de una variable, depende del tipo de valor que vaya a almacenar. Distinguimos varios tipos de valores que podemos clasificar en: tipos enteros, **short**, **int**, **long** y **char**, tipos reales, **float** y **double** y el tipo **bool**.

Cada tipo tiene un rango diferente de valores positivos y negativos, excepto el **bool** que sólo tiene dos valores: **true** y **false**. Por lo tanto, el tipo que se seleccio-

ne para declarar cada variable en un determinado programa dependerá del rango y tipo de los valores que vayan a almacenar: enteros, fraccionarios o booleanos.

El tipo **short** permite declarar datos enteros comprendidos entre -32768 y $+32767$ (16 bits de longitud), el tipo **int** declara datos enteros comprendidos entre -2147483648 y $+2147483647$ (32 bits de longitud) y el tipo **long** dependiendo de la implementación puede ser de 32 bits de longitud, igual que un **int** (es nuestro caso), o de 64 bits (-9223372036854775808 a $+9223372036854775807$). A continuación se muestran algunos ejemplos:

```
short i = 0, j = 758; // short es sinónimo de short int
int k = -125000000;
long l = 125000000; // long es sinónimo de long int
```

A los tipos anteriores, C++ añade los correspondientes tipos enteros sin signo: **unsigned short** (0 a $2^{16}-1$), **unsigned int** (0 a $2^{32}-1$) y **unsigned long** (0 a $2^{32}-1$).

El tipo **char** es utilizado para declarar datos enteros en el rango -128 a 127 . Los valores 0 a 127 se corresponden con los caracteres ASCII del mismo código (ver los apéndices). El juego de caracteres ASCII conforma una parte muy pequeña del juego de caracteres Unicode (valores de 0 a 65535), donde cada carácter ocupa 16 bits. Hay lenguajes que utilizan este código con el único propósito de internacionalizar el lenguaje. C++ cubre este objetivo con el tipo **wchar_t**.

El siguiente ejemplo declara la variable *car* de tipo **char** a la que se le asigna el carácter 'a' como valor inicial (observe que hay una diferencia entre 'a' y a; a entre comillas simples es interpretada por el compilador C++ como un valor, un carácter, y a sin comillas sería interpretada como una variable). Las dos declaraciones siguientes son idénticas:

```
char car = 'a';
char car = 97; // la 'a' es el decimal 97
```

El tipo **float** (32 bits de longitud) se utiliza para declarar un dato que puede contener una parte decimal. Los datos de tipo **float** almacenan valores con una precisión aproximada de 6 dígitos. Por ejemplo:

```
float a = 3.14159F;
float b = 2.2e-5F; // 2.2e-5 = 2.2 por 10 elevado a -5
float c = 2.0/3.0F; // 0,666667
```

Para especificar que una constante fraccionaria (no entera) es de tipo **float**, hay que añadir al final de su valor la letra 'f' o 'F', de lo contrario será considerada de tipo **double**.

El tipo **double** (64 bits de longitud) se utiliza para declarar un dato que puede contener una parte decimal. Los datos de tipo **double** almacenan valores con una precisión aproximada de 15 dígitos. El siguiente ejemplo declara la variable *a*, de tipo real de precisión doble:

```
double a = 3.14159; // una constante es double por omisión
```

ASIGNAR VALORES

La finalidad de un programa es procesar datos *numéricos* y *cadena de caracteres* para obtener un resultado. Estos datos, generalmente, estarán almacenados en variables y el resultado obtenido también será almacenado en variables. ¿Cómo son almacenados? Pues a través de las funciones proporcionadas por la biblioteca de C++, o bien utilizando una sentencia de asignación de la forma:

variable operador_de_asignación valor

Una sentencia de asignación es asimétrica. Esto quiere decir que se evalúa la expresión que está a la derecha del operador de asignación y el resultado se asigna a la variable especificada a su izquierda. Por ejemplo:

```
resultado = dato1 + dato2; // + es el operador de asignación
```

Pero, según lo expuesto, no sería válido escribir:

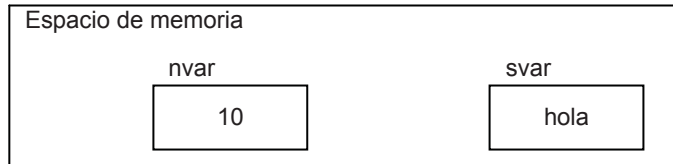
```
dato1 + dato2 = resultado;
```

Mientras que los datos numéricos son almacenados en variables de alguno de los tipos de valores expuestos anteriormente, las cadenas de caracteres son almacenadas en objetos de la clase **string** o en matrices, cuyo estudio se pospone para un capítulo posterior; no obstante veamos un ejemplo. Un objeto de la clase **string** (su manipulación implica incluir el fichero de cabecera `<string>`) se define y se le asigna un valor así:

```
string cadena; // cadena es un objeto string
cadena = "hola"; // asignar a cadena la constante "hola"
// Ambas sentencias equivalen a: string cadena = "hola";
```

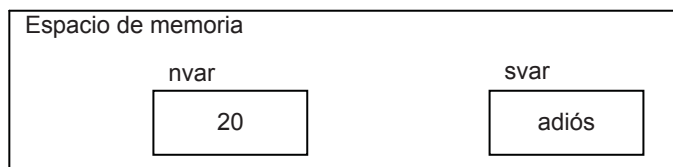
Cuando se asigna un valor a una variable estamos colocando ese valor en una localización de memoria asociada con esa variable. Por ejemplo:

```
int nvar = 10; // variable de un tipo entero (int)
string svar = "hola"; // objeto de tipo string
```



Lógicamente, cuando la variable tiene asignado un valor y se le asigna uno nuevo, el valor anterior es destruido ya que el valor nuevo pasa a ocupar la misma localización de memoria. En el ejemplo siguiente, se puede observar, con respecto a la situación anterior, que el contenido de *nvar* se modifica con un nuevo valor 20, y que el objeto *svar* también se modifica; ahora el objeto *svar* de tipo **string** contiene “adiós”.

```
nvar = 20;
svar = "adiós";
```



El siguiente ejemplo declara tres variables numéricas *a*, *b* y *c*, y un objeto **string** *s*; después asigna valores a esas variables y al objeto.

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

int main()
{
    double a = 0, b = 0;
    int c = 0;
    string s;

    a = 3.14; b = 2.71; c = 2;
    s = "Datos";
}
```

AÑADIR COMENTARIOS

Un comentario es un mensaje a cualquiera que lea el código fuente. Añadiendo comentarios se hace más fácil la comprensión de un programa. La finalidad de los comentarios es explicar el código fuente. Se pueden utilizar comentarios acotados o de una sola línea.

Un comentario acotado empieza con los caracteres `/*` y finaliza con los caracteres `*/`. Estos comentarios pueden ocupar más de una línea, pero no pueden anidarse. Por ejemplo:

```
/*
 * Asignar datos:
 *   a, b, c representan datos numéricos.
 *   s representa una cadena de caracteres.
 */

int main()
{
    double a = 0, b = 0;
    int c = 0;
    string s;

    a = 3.14; b = 2.71; c = 2;
    s = "Datos";
}
```

Un comentario de una sola línea comienza con una doble barra (`//`) y se extiende hasta el final de la línea. Por ejemplo, el siguiente programa declara tres variables numéricas *a*, *b* y *c*, y un objeto *s* de tipo **string** (cadena de caracteres); después asigna valores a las variables y al objeto, y finalmente los muestra.

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

// Asignar datos:
//   a, b, c representan datos numéricos.
//   s representa una cadena de caracteres.
int main()
{
    double a = 0, b = 0;
    int c = 0;
    string s;

    a = 3.14; b = 2.71; c = 2;
    s = "Datos";

    cout << s + ":\n";
    cout << " a = " << a << '\n';
    cout << " b = " << b << '\n';
    cout << " c = " << c << '\n';
}
```

Ejecución del programa:

```
Datos:
a = 3,14
b = 2,71
c = 2
```

La constante de carácter (carácter encerrado entre comillas simples) `\n` especifica un salto al principio de la línea siguiente.

EXPRESIONES ARITMÉTICAS

Una expresión es un conjunto de operandos unidos mediante operadores para especificar una operación determinada. Todas las expresiones cuando se evalúan retornan un valor. Por ejemplo, la siguiente expresión retorna la suma de `dato1` y `dato2`:

```
dato1 + dato2
```

C++ define cinco operadores aritméticos que son los siguientes:

- + *Suma*. Los operandos pueden ser enteros o reales.
- *Resta*. Los operandos pueden ser enteros o reales.
- * *Multipliación*. Los operandos pueden ser enteros o reales.
- / *División*. Los operandos pueden ser enteros o reales. Si ambos operandos son enteros, el resultado es entero. En el resto de los casos el resultado es real.
- % *Módulo* o resto de una división entera. Los operandos tienen que ser enteros.

Cuando en una operación aritmética los operandos son de diferentes tipos, ambos son convertidos al tipo del operando de precisión más alta. En una asignación, el resultado obtenido en una operación aritmética es convertido implícita o explícitamente al tipo de la variable que almacena dicho resultado (véase *Conversión entre tipos primitivos* en el capítulo *Elementos del lenguaje C++*).

Así mismo, cuando en una expresión intervienen varios operadores aritméticos, estos se ejecutan de izquierda a derecha y de mayor a menor prioridad. Los operadores `*`, `/` y `%` tienen entre ellos la misma prioridad pero mayor que la de los operadores `+` y `–` que también tienen la misma prioridad entre ellos. Una expresión entre paréntesis siempre se evalúa primero; si hay varios niveles de paréntesis, son evaluados de más internos a más externos. Por ejemplo:

```
#include <iostream>
#include <cmath>

using namespace std;
```

```

/*
 * Operaciones aritméticas
 */
int main()
{
    double a = 10;
    float b = 20;
    int c = 2, r = 0;
    r = static_cast<int>(7.5 * sqrt(a) - b / c);
    cout << r << '\n';
}

```

Ejecución del programa:

13

En este ejemplo, primero se realiza la operación $\text{sqrt}(a)$ (invoca a la función **sqrt** de la biblioteca de C++ para calcular la raíz cuadrada de a ; esto requiere incluir el fichero de cabecera `<cmath>`) y después, el resultado de tipo **double** que se obtiene se multiplica por 7,5. A continuación se realiza b / c convirtiendo previamente c al tipo de b ; el resultado que se obtiene es de tipo **float**. Finalmente se hace la resta de los dos resultados anteriores convirtiendo previamente el resultado de tipo **float** a tipo **double**; se obtiene un resultado de tipo **double** que, como puede observar, es convertido explícitamente a tipo **int** (`static_cast<int>`), truncando la parte decimal, para poder almacenarlo en r . Si no se realiza una conversión explícita, el compilador realizará una conversión implícita **double** a **int** y como esta operación acarrea una pérdida de precisión, avisará de ello al programador por si tal hecho hubiera pasado desapercibido.

EXPRESIONES CONDICIONALES

En ocasiones interesará dirigir el flujo de ejecución de un programa por un camino u otro en función del valor de una expresión. Para ello, C++ proporciona la sentencia **if**. Para ver cómo se utiliza esta sentencia, vamos a realizar un ejemplo que verifique si un número es par. En caso afirmativo imprimirá un mensaje “Número par” y a continuación el valor del número. En caso negativo sólo imprimirá el valor del número:

```

#include <iostream>
using namespace std;
/*
 * Expresiones condicionales
 */
int main()
{
    int num = 24;

```

```

if ( num % 2 == 0 ) // si el resto de la división es igual a 0,
    cout << "Número par\n";

cout << "Valor: " << num << '\n';
}

```

La sentencia **if** del ejemplo anterior se interpreta así: si la condición especificada entre paréntesis, $num \% 2 == 0$, es cierta, invocar a **cout** y escribir “Número par”; si es falsa, no hacer lo anterior. En cualquiera de los dos casos, continuar con la siguiente sentencia ($cout << "Valor: " << num << '\n'$). Según esto, el resultado después de ejecutar este programa será:

```

Número par
Valor: 24

```

Si el número hubiera sido 23, el resultado hubiera sido sólo *Valor: 23*. La expresión que hay entre paréntesis a continuación de **if** es una *expresión condicional* y el resultado de su evaluación siempre es un valor booleano **true** (verdadero) o **false** (falso); estas dos constantes están predefinidas en C++. Los operadores de relación o de comparación que podemos utilizar en estas expresiones son los siguientes:

```

<   ¿Primer operando menor que el segundo?
>   ¿Primer operando mayor que el segundo?
<=  ¿Primer operando menor o igual que el segundo?
>=  ¿Primer operando mayor o igual que el segundo?
!=   ¿Primer operando distinto que el segundo?
==   ¿Primer operando igual que el segundo?

```

Modifiquemos el programa anterior para que ahora indique si el número es par o impar. Para este caso emplearemos una segunda forma de la sentencia **if** que consiste en añadir a la anterior la cláusula **else** (si no):

```

#include <iostream>
using namespace std;
/*
 * Expresiones condicionales
 */
int main()
{
    int num = 23;

    if ( num % 2 == 0 ) // si el resto de la división es igual a 0,
        cout << "Número par\n";
}

```

```

else // si el resto de la división no es igual a 0,
  cout << "Número impar\n";

  cout << "Valor: " << num << '\n';
}

```

Ejecución del programa:

```

Número impar
Valor: 23

```

La sentencia **if** de este otro ejemplo se interpreta así: si la condición especificada entre paréntesis, $num \% 2 == 0$, es cierta, invocar a **cout** y escribir “Número par” y si no, invocar a **cout** y escribir “Número impar”. En cualquiera de los dos casos continuar con la siguiente sentencia del programa.

A continuación se muestra otra versión del programa anterior que produciría exactamente los mismos resultados. No obstante, representa un estilo peor de programación, ya que repite código, que como hemos visto, se puede evitar.

```

#include <iostream>
using namespace std;
/*
 * Expresiones condicionales
 */
int main()
{
  int num = 23;

  if ( num % 2 == 0) // si el resto de la división es igual a 0,
  {
    cout << "Número par" << '\n';
    cout << "Valor: " << num << '\n';
  }
  else // si el resto de la división no es igual a 0,
  {
    cout << "Número impar" << '\n';
    cout << "Valor: " << num << '\n';
  }
}

```

Se puede observar un nuevo detalle, y es que cuando el número de sentencias que se desean ejecutar en función del resultado **true** o **false** de una expresión es superior a una, hay que encerrarlas en un bloque. En el capítulo *Sentencias de control* veremos la sentencia **if** con más detalle.

Observe que los operadores de asignación (=) y de igualdad (==) son diferentes.

ESCRIBIR NUESTRAS PROPIAS FUNCIONES

De la misma forma que la biblioteca de C++ proporciona funciones predefinidas como `sqrt`, nosotros también podemos añadir a nuestro programa nuestras propias funciones e invocarlas de la misma forma que lo hacemos con las predefinidas.

Por ejemplo, en el programa siguiente la función `main` muestra la suma de dos valores cualesquiera; dicha suma la obtiene invocando a una función `sumar`, añadida por nosotros, que recibe en sus parámetros `x` e `y` los valores a sumar, realiza la suma de ambos `y`, utilizando la sentencia `return`, devuelve el resultado solicitado por `main`.

```

double sumar(double x, double y)
{
    double resultado = 0;
    // Realizar cálculos
    return resultado;
}

```

Tipo del valor retornado

Parámetros que se pasarán como argumentos

Valor retornado por la función sumar

Han aparecido algunos conceptos nuevos (argumentos pasados a una función y valor retornado por una función). No se preocupe, sólo se trata de un primer contacto. Más adelante estudiaremos todo esto con mayor profundidad. Para una mejor comprensión de lo dicho, piense en la función llamada *logaritmo* que seguro habrá utilizado más de una vez a lo largo de sus estudios. Esta función devuelve un valor real correspondiente al logaritmo del valor pasado como argumento: $x = \log(y)$. Bueno, pues compárela con la función `sumar` y comprobará que estamos hablando de cosas análogas.

Evidentemente, se pueden escribir funciones que no requieran devolver un valor, lo que se indicará mediante la palabra reservada `void`. Por ejemplo:

```

void mensaje() // esta función no devuelve nada
{
    cout << "un mensaje\n";
}

```

Según lo expuesto y aplicando los conocimientos adquiridos hasta ahora, el programa propuesto puede ser como se muestra a continuación:

```

#include <iostream>

using namespace std;

```



```

/*
 * Función sumar:
 *  parámetros x e y de tipo double
 *  devuelve x + y
 */

double sumar(double x, double y)
{
    double resultado = 0;
    resultado = x + y;
    return resultado;
}

int main()
{
    double a = 10, b = 20, r = 0;
    r = sumar(a, b);
    cout << "Suma = " << r << '\n';
}

```

Ejecución del programa:

Suma = 30

Observe cómo es la llamada a la función *sumar*: $r = \text{sumar}(a, b)$. La función es invocada por su nombre, entre paréntesis se especifican los argumentos con los que debe operar, y el resultado que devuelve se almacena en *r*.

Finalmente, si comparamos el esqueleto de la función *sumar* y el de la función **main**, observamos que son muy parecidos: *sumar* devuelve un valor de tipo **double** y **main** un entero (eso es lo que indica **int**) y *sumar* tiene dos parámetros, *x* e *y*, y **main** ninguno, en este caso.

EJERCICIOS PROPUESTOS

1. Escriba una aplicación que visualice en el monitor los siguientes mensajes:

```

Bienvenido al mundo de C++.
Podrás dar solución a muchos problemas.

```

2. Decida qué tipos de valores necesita para escribir un programa que calcule la suma y la media de cuatro números de tipo **int**. Escriba un programa como ejemplo.
3. Escriba un programa que incluya una función denominada *calcular* que devuelva como resultado el valor de la expresión:

$$\frac{b^2 - 4ac}{2a}$$

La función **main** invocará a *calcular* pasando los valores de $a = 1$, $b = 5$ y $c = 2$ y mostrará el resultado obtenido.

ELEMENTOS DEL LENGUAJE C++

En este capítulo veremos los elementos que aporta C++ (caracteres, secuencias de escape, tipos de datos, operadores, etc.) para escribir un programa. El introducir este capítulo ahora es porque dichos elementos los tenemos que utilizar desde el principio; algunos ya han aparecido en los ejemplos de los capítulos anteriores. Por lo tanto, considere este capítulo como soporte para el resto de los capítulos; esto es, lo que se va a exponer en él lo irá utilizando en menor o mayor medida en los capítulos sucesivos. Por lo tanto, límitese ahora simplemente a realizar un estudio para saber de forma genérica los elementos con los que contamos para desarrollar nuestros programas.

PRESENTACIÓN DE LA SINTAXIS DE C++

Las palabras clave aparecerán en negrita y cuando se utilicen deben escribirse exactamente como aparecen. Por ejemplo:

```
char a;
```

El texto que aparece en cursiva significa que ahí debe ponerse la información indicada por ese texto. Por ejemplo:

```
typedef declaración_tipo sinónimo[, sinónimo]...;
```

Una información encerrada entre corchetes "[]" es opcional. Los puntos suspensivos "..." indican que pueden aparecer más elementos de la misma forma.

Cuando dos o más opciones aparecen entre llaves "{}" separadas por "|", se elige una, la necesaria dentro de la sentencia. Por ejemplo:

```
constante_entera[{L|U|UL}]
```

CARACTERES DE C++

Los caracteres de C++ pueden agruparse en letras, dígitos, espacios en blanco, caracteres especiales, signos de puntuación y secuencias de escape.

Letras, dígitos y carácter de subrayado

Estos caracteres son utilizados para formar las *constantes*, los *identificadores* y las *palabras clave* de C++. Son los siguientes:

- Letras mayúsculas del alfabeto inglés:
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 Z
- Letras minúsculas del alfabeto inglés:
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 z
- Dígitos decimales:
0 1 2 3 4 5 6 7 8 9
- Carácter de subrayado "_"

El compilador C++ trata las letras mayúsculas y minúsculas como caracteres diferentes. Por ejemplo, los identificadores *Pi* y *PI* son diferentes.

Espacios en blanco

Los caracteres espacio en blanco, tabulador horizontal, tabulador vertical, avance de página y nueva línea son caracteres denominados *espacios en blanco*, porque la labor que desempeñan es la misma que la del espacio en blanco: actuar como separadores entre los elementos de un programa, lo cual permite escribir programas más legibles. Por ejemplo, el siguiente código:

```
int main() { cout << "Hola, qué tal estáis.\n"; }
```

puede escribirse de una forma más legible así:

```
int main()
{
    cout << "Hola, qué tal estáis.\n";
}
```

Los espacios en blanco en exceso son ignorados por el compilador. Según esto, el código siguiente se comporta exactamente igual que el anterior:

```
int main()
{
    cout << "Hola, qué tal estáis.\n";
}
```

← línea en blanco

← espacios en blanco

La secuencia *Ctrl+Z* en Windows o *Ctrl+D* en Linux es tratada por el compilador como un indicador de fin de fichero (*End Of File*).

Caracteres especiales y signos de puntuación

Este grupo de caracteres se utiliza de diferentes formas; por ejemplo, para indicar que un identificador es una función o una matriz; para especificar una determinada operación aritmética, lógica o de relación; etc. Son los siguientes:

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

Secuencias de escape

Cualquier carácter de los anteriores puede también ser representado por una *secuencia de escape*. Una secuencia de escape está formada por el carácter `\` seguido de una *letra* o de una *combinación de dígitos*. Son utilizadas para acciones como nueva línea, tabular y para hacer referencia a caracteres no imprimibles.

El lenguaje C++ tiene predefinidas las siguientes secuencias de escape:

Secuencia	Nombre
<code>\n</code>	Ir al principio de la siguiente línea.
<code>\t</code>	Tabulador horizontal.
<code>\v</code>	Tabulador vertical (sólo para la impresora).
<code>\b</code>	Retroceso (<i>backspace</i>).
<code>\r</code>	Retorno de carro sin avance de línea.
<code>\f</code>	Alimentación de página (sólo para la impresora).
<code>\a</code>	Alerta, pitido.
<code>\'</code>	Comilla simple.
<code>\"</code>	Comilla doble.
<code>\\</code>	Barra invertida (<i>backslash</i>).
<code>\ddd</code>	Carácter ASCII. Representación octal (<i>d</i> es un dígito del 0 al 7).
<code>\xdd</code>	Carácter ASCII. Representación hexadecimal (<i>d</i> es un dígito del 0 al 9 o una letra <i>A - Z</i> o <i>a - z</i>).

Observe en la llamada a `cout` del ejemplo anterior la secuencia de escape `\n`.

TIPOS DE DATOS

Recuerde las operaciones aritméticas que realizaba el proyecto *Aritmetica* que vimos en el capítulo 1. Por ejemplo, una de las operaciones que realizábamos era la suma de dos valores:

```
dato1 = 20;
dato2 = 10;
resultado = dato1 + dato2;
```

Para que el compilador C++ reconozca esta operación es necesario especificar previamente el tipo de cada uno de los operandos que intervienen en la misma, así como el tipo del resultado. Para ello, escribiremos una línea como la siguiente:

```
int dato1, dato2, resultado;
dato1 = 20;
dato2 = 10;
resultado = dato1 + dato2;
```

La declaración anterior le indica al compilador C++ que *dato1*, *dato2* y *resultado* son de tipo entero (**int**). Observe que se puede declarar más de una variable del mismo tipo utilizando una lista separada por comas.

Los tipos de datos en C++ se clasifican en: tipos *primitivos* y tipos *derivados*. La razón para ofrecer más de un tipo de datos es permitir al programador aprovechar las características del hardware, ya que diferentes máquinas pueden presentar diferencias significativas en los requerimientos de memoria, tiempo de acceso a memoria y velocidad de cálculo entre los distintos tipos.

Tipos primitivos

Se les llama primitivos porque están definidos por el compilador y se clasifican en: tipos enteros, tipos reales, el tipo carácter ampliado **wchar_t** y el tipo **bool**.

Tipos enteros: **char**, **short**, **int** y **long**.

Tipos reales: **float**, **double** y **long double**.

Cada tipo primitivo tiene un rango diferente de valores positivos y negativos, excepto el tipo **bool** que sólo tiene dos valores: **true** y **false**. El tipo de datos que se seleccione para declarar las variables de un determinado programa dependerá del rango y tipo de valores que vayan a almacenar cada una de ellas y de si éstos son enteros o fraccionarios. Los ficheros de cabecera `<limits>` y `<float>`, así como la plantilla `numeric_limits<tipo>` definida en `<limits>`, especifican los valores máximo y mínimo para cada tipo, además de otras características.

Cada tipo entero puede ser calificado por las palabras clave **signed** o **unsigned**. Un entero calificado **signed** es un entero con signo; esto es, un valor entero positivo o negativo. Un entero calificado **unsigned** es un valor entero sin signo, el cual es manipulado como un valor entero positivo. Esta calificación da lugar a los siguientes tipos extras:

```
signed char,   unsigned char
signed short, unsigned short
signed int,    unsigned int
signed long,   unsigned long
```

Si los calificadores **signed** y **unsigned** se utilizan sin un tipo entero específico, se asume el tipo **int**. Por este motivo, las siguientes declaraciones son equivalentes:

```
signed x;           // es equivalente a
signed int x;
```

Un tipo entero calificado con **signed** es equivalente a utilizarlo sin calificar. Según esto, las siguientes declaraciones son equivalentes:

```
char y;             // es equivalente a
signed char y;
```

Los tipos enteros y reales (excepto el tipo **long double**) ya fueron comentados en el capítulo 1. El tipo **long double** dependiendo de la implementación C++ utilizada puede ser de 8 bytes de longitud, igual que un **double**, o de 12 bytes.

El tipo **wchar_t** es utilizado para declarar datos enteros en el rango 0 a 65535 lo que permite manipular el juego de caracteres Unicode. Se trata de un código de 16 bits (valores de 0 a 65535), esto es, cada carácter ocupa 2 bytes, con el único propósito de internacionalizar el lenguaje. Los valores 0 a 127 se corresponden con los caracteres ASCII o ANSI del mismo código (ver los apéndices).

El tipo **bool** se utiliza para indicar si el resultado de la evaluación de una expresión booleana es verdadero o falso. Por definición, **true** toma el valor 1 cuando se convierte a entero y **false** el valor 0, y a la inversa, cualquier valor entero distinto de cero se convierte en **true** y cero en **false**.

Tipos derivados

Se les denomina tipos derivados porque se construyen a partir de los tipos primitivos y se pueden clasificar en: enumeraciones, matrices, funciones, punteros, re-

ferencias, estructuras, uniones y clases. Todos ellos serán explicados en éste y en sucesivos capítulos.

Enumeraciones

Crear una enumeración supone definir un nuevo tipo de datos y declarar una variable de ese tipo. La sintaxis es la siguiente:

```
enum enumeración
{
    // constantes enteras que forman la enumeración
};
```

donde *enumeración* es un identificador que nombra el nuevo tipo definido.

Después de definir una enumeración, podemos declarar una o más variables de ese tipo, de la forma:

```
[enum] enumeración variable[, variable]....;
```

El siguiente ejemplo declara una variable llamada *color* del tipo enumerado *colores*, la cual puede tomar cualquier valor de los especificados en la lista, por ejemplo *amarillo*.

```
enum colores
{
    azul, amarillo, rojo, verde, blanco, negro
};

colores color;

color = amarillo;
```

Cada identificador de la lista de constantes en una enumeración se corresponde con un valor entero, de tal forma que, por defecto, el primer identificador se corresponde con el valor 0, el siguiente con el valor 1, y así sucesivamente. No obstante, para C++ un tipo enumerado es un nuevo tipo entero diferente de los anteriores. Esto significa que en C++ un valor de tipo **int** no puede ser asignado directamente a una variable de un tipo enumerado, sino que hay que hacer una conversión explícita de tipo (véase *Conversión entre tipos de datos* al final de este capítulo). La conversión inversa sí es posible. Por ejemplo:

```
color = 3; // error: conversión implícita int-colores no permitida.
color = static_cast<colores>(3); // correcto: conversión explícita
// de int a colores.
int c = verde; // correcto.
```


A cualquier identificador de la lista se le puede asignar un valor inicial entero por medio de una expresión constante. Los identificadores sucesivos tomarán valores correlativos a partir de éste. Por ejemplo:

```
enum colores
{
    azul, amarillo, rojo, verde = 0, blanco, negro
} color;
```

Este ejemplo define un tipo enumerado llamado *colores* y declara una variable *color* de ese tipo. Los valores asociados a los identificadores son los siguientes: *azul* = 0, *amarillo* = 1, *rojo* = 2, *verde* = 0, *blanco* = 1 y *negro* = 2.

A las enumeraciones se les aplica las siguientes reglas:

- Dos o más miembros de una enumeración pueden tener un mismo valor.
- Un mismo identificador no puede aparecer en más de una enumeración.
- Desafortunadamente, no es posible leer o escribir directamente un valor de un tipo enumerado. El siguiente ejemplo aclara este detalle.

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

enum colores
{
    azul, amarillo, rojo, verde, blanco, negro
};

int main()
{
    colores color;
    // Leer un color introducido desde el teclado
    cout << "Color: ";
    // cin >> color: sentencia de lectura no permitida. La
    // sustituimos por las tres siguientes:
    int ncolor;
    cin >> ncolor; // leer un color: 0, 1, 2, etc.
    color = static_cast<colores>(ncolor); // conversión explícita
    // Visualizar un color
    cout << color << '\n';
}
```

Ejecución del programa:

```
Color: 3[Entrar]
3
```

En un próximo capítulo verá con detalle el objeto **cin**; ahora límitese a saber que este objeto le permite asignar un valor introducido por el teclado a la variable especificada. En el ejemplo anterior se observa que no es posible asignar a la variable *color* directamente a través del teclado una constante de la enumeración, por ejemplo *verde*, sino que hay que hacerlo indirectamente leyendo la constante entera equivalente (en el ejemplo, 3). Igualmente, **cout** no escribirá *verde*, sino que escribirá 3. Según esto, se preguntará: ¿qué aportan, entonces, las enumeraciones? Las enumeraciones ayudan a acercar más el lenguaje de alto nivel a nuestra forma de expresarnos. Como podrá ver más adelante, la expresión “si el color es verde,...” dice más que la expresión “si el color es 3,...”.

Clases

El lenguaje C++ es un lenguaje orientado a objetos. La base de la programación orientada a objetos es la *clase*. Una clase es un tipo de objetos definido por el usuario. Por ejemplo, la clase **string** de la biblioteca C++ está definida así:

```
class string
{
    // Atributos
    // Métodos
}
```

Y, ¿cómo se define un objeto de esta clase? Pues, una forma de hacerlo sería así:

```
string sTexto = "abc";
```

Suponiendo que la clase **string** tiene un operador de indexación de acceso público, `[i]`, que devuelve el carácter que está en la posición *i*, la siguiente sentencia devolverá el carácter que está en la posición 1 (la ‘b’):

```
char car = sTexto[1];
```

Una característica muy importante que aporta la programación orientada a objetos es la *herencia* ya que permite la reutilización del código escrito por nosotros o por otros. Por ejemplo, el siguiente código define la clase *ofstream* como una clase derivada (que hereda) de *ostream*:

```
class ofstream : ostream
{
    // Atributos
    // Métodos
}
```

La clase *ofstream* incluirá los atributos y métodos heredados de *ostream* más los atributos y métodos que se hayan definido en esta clase. Esto significa que un objeto de la clase *ofstream* podrá ser manipulado por los métodos heredados y por los propios.

SINÓNIMOS DE UN TIPO

Utilizando la palabra reservada **typedef** podemos declarar nuevos nombres de tipos de datos; esto es, sinónimos de otro tipo ya sean primitivos o derivados, los cuales pueden ser utilizados más tarde para declarar variables de esos tipos. La sintaxis de **typedef** es la siguiente:

```
typedef declaración_tipo sinónimo[, sinónimo]...;
```

donde *declaración_tipo* es cualquier tipo definido en C++, primitivo o derivado, y *sinónimo* es el nuevo nombre elegido para el tipo especificado.

Por ejemplo, la sentencia siguiente declara el nuevo tipo *ulong* como sinónimo del tipo primitivo **unsigned long**:

```
typedef unsigned long ulong;
```

Una vez definido el tipo *ulong* como sinónimo de **unsigned long**, sería posible declarar una variable *dni* de cualquiera de las dos formas siguientes:

```
unsigned long dni; // o bien
ulong dni;
```

Las declaraciones **typedef** permiten parametrizar un programa para evitar problemas de portabilidad. Si utilizamos **typedef** con los tipos que pueden depender de la instalación, cuando se lleve el programa a otra instalación sólo se tendrán que cambiar estas declaraciones. Por ejemplo:

```
typedef long int32;
```

Si ahora se porta el programa que contiene esta declaración a otra plataforma donde **long** tiene una longitud de 64 bits en lugar de 32, bastaría con cambiar la sentencia anterior por esta otra, suponiendo que la longitud de un **int** son 32 bits:

```
typedef int int32;
```

LITERALES

Un literal en C++ puede ser: un entero, un real, un valor booleano, un carácter, una cadena de caracteres y una constante como **NULL**. Por ejemplo, son literales: *5*, *3.14*, **true**, *'a'*, *"hola"*. En realidad son valores constantes.

Literales enteros

El lenguaje C++ permite especificar un literal entero en base 10, 8 y 16.

En general, el signo + es opcional si el valor es positivo y el signo – estará presente siempre que el valor sea negativo. El tipo de un literal entero depende de su base, de su valor y de su sufijo. La sintaxis para especificar un literal entero es:

```
{[+]|-}literal_entero[ {L|U|UL} ]
```

Si el literal es decimal y no tiene sufijo, su tipo es el primero de los tipos **int**, **long int** o **unsigned long int** en el que su valor pueda ser representado.

Si es octal o hexadecimal y no tiene sufijo, su tipo es el primero de los tipos **int**, **unsigned int**, **long int** o **unsigned long int** en el que su valor pueda ser representado.

También se puede indicar explícitamente el tipo de un literal entero, añadiendo los sufijos *L*, *U* o *UL* (mayúsculas o minúsculas).

Si el sufijo es *L*, su tipo es **long** cuando el valor puede ser representado en este tipo; si no, es **unsigned long**. Si el sufijo es *U*, su tipo es **unsigned int** cuando el valor puede ser representado en este tipo; si no, es **unsigned long**. Si el sufijo es *UL*, su tipo es **unsigned long**.

Un *literal entero decimal* puede tener uno o más dígitos del 0 a 9, de los cuales el primero de ellos es distinto de 0. Por ejemplo:

```
4326      constante entera int
1522U     constante entera unsigned int
1000L     constante entera long
325UL     constante entera unsigned long
```

Un *literal entero octal* puede tener uno o más dígitos del 0 a 7, precedidos por 0 (cero). Por ejemplo:

```
0326      constante entera int en base 8
```

Un *literal entero hexadecimal* puede tener uno o más dígitos del 0 a 9 y letras de la *A* a la *F* (en mayúsculas o en minúsculas) precedidos por *0x* o *0X* (*cero* seguido de *x*). Por ejemplo:

```
256      número decimal 256
0400     número decimal 256 expresado en octal
0x100    número decimal 256 expresado en hexadecimal
```

-0400 número decimal -256 expresado en octal
 -0x100 número decimal -256 expresado en hexadecimal

Literales reales

Un literal real está formado por una *parte entera*, seguida por un *punto decimal*, y una *parte fraccionaria*. También se permite la notación científica, en cuyo caso se añade al valor una *e* o *E*, seguida por un exponente positivo o negativo.

`{[+]|-}parte-entera.parte-fraccionaria[{e|E} {[+]|-}exponente]`

donde *exponente* representa cero o más dígitos del 0 al 9 y *E* o *e* es el símbolo de exponente de la base 10 que puede ser positivo o negativo ($2E-5 = 2 \times 10^{-5}$). Si la constante real es positiva, no es necesario especificar el signo y si es negativa lleva el signo menos (-). Por ejemplo:

-17.24
 17.244283
 .008e3
 27E-3

Un literal real tiene siempre tipo **double**, a no ser que se añada al mismo una *f* o *F*, en cuyo caso será de tipo **float**. Por ejemplo:

17.24F constante real de tipo float

Literales de un solo carácter

Los literales de un solo carácter son de tipo **char**. Este tipo de literales está formado por un único carácter encerrado entre *comillas simples*. Una secuencia de escape es considerada como un único carácter. Algunos ejemplos son:

' ' espacio en blanco
 'x' letra minúscula x
 '\n' retorno de carro más avance de línea
 '\x07' pitido
 '\x1B' carácter ASCII Esc

El valor de una constante de un solo carácter es el valor que le corresponde en el juego de caracteres de la máquina.

Los literales de un solo carácter ampliado (**wchar_t**) son de la forma *Lx'*. Por ejemplo:

```
wchar_t wcar = L'a'; // el tamaño de wcar es 2 bytes
```

Literales de cadenas de caracteres

Un literal de cadena de caracteres es una secuencia de caracteres encerrados entre *comillas dobles* (incluidas las secuencias de escape como `\`). Por ejemplo:

```
"Esto es una constante de caracteres"  
"3.1415926"  
"Paseo de Pereda 10, Santander"  
"" // cadena vacía  
"Lenguaje \"C++\" // produce: Lenguaje "C++"
```

IDENTIFICADORES

Los identificadores son nombres dados a tipos, literales, variables, funciones, etiquetas de un programa, etc. La sintaxis para formar un identificador es:

$$\{ \text{letra} | _ \} [\{ \text{letra} | \text{dígito} | _ \}] \dots$$

lo cual indica que un identificador consta de uno o más caracteres (véase el apartado anterior *Letras, dígitos y carácter de subrayado*) y que el *primer carácter* debe ser una *letra* o el *carácter de subrayado*. No pueden comenzar por un dígito ni pueden contener caracteres especiales (véase el apartado anterior *Caracteres especiales y signos de puntuación*).

Los identificadores pueden tener cualquier número de caracteres pero dependiendo del compilador que se utilice (en particular, del enlazador), solamente los n primeros caracteres son significativos. Esto quiere decir que un identificador es distinto de otro cuando difieren al menos en uno de los n primeros caracteres significativos. Algunos ejemplos son:

```
Suma  
suma  
Calculo_Numeros_Primos  
ordenar  
VisualizarDatos
```

PALABRAS CLAVE

Las palabras clave son identificadores predefinidos que tienen un significado especial para el compilador C++. Por lo tanto, un identificador definido por el usua-

rio, no puede tener el mismo nombre que una palabra clave. Algunas de las palabras clave que utiliza el lenguaje C++ son las siguientes:

and	do	int	short	typeid
auto	double	long	signed	union
bool	else	namespace	sizeof	unsigned
break	enum	new	static	using
case	extern	not	struct	virtual
catch	false	operator	switch	void
char	float	or	template	wchar_t
class	for	private	this	while
const	friend	protected	throw	xor
continue	goto	public	true	
default	if	register	try	
delete	inline	return	typedef	

Las palabras clave deben escribirse siempre en minúsculas, como están.

DECLARACIÓN DE CONSTANTES SIMBÓLICAS

Declarar una constante simbólica significa decirle al compilador C++ el nombre de la constante y su valor. La sintaxis para declarar una constante es así:

```
const tipo nombre = valor
```

El siguiente ejemplo declara la constante real *PI* con el valor 3.14159, la constante de un solo carácter *NL* con el valor ‘\n’ y la constante de caracteres *MENSAJE* con el valor “Pulse una tecla para continuar\n”:

```
const double PI = 3.14159;
const char NL = '\n';
const string MENSAJE = "Pulse una tecla para continuar\n";
```

El escribir los nombres de las constantes en mayúsculas para diferenciarlas de las variables es sólo una cuestión de estilo.

Una vez declarada e iniciada una constante, ya no se puede modificar su valor; por eso se inicia al declararla. Por ejemplo, suponiendo declarada la constante *PI*, la siguiente sentencia daría lugar a un error:

```
PI = 3.1416; // error
```

¿Por qué utilizar constantes?

Utilizando constantes es más fácil modificar un programa. Por ejemplo, supongamos que un programa utiliza N veces una constante de valor 3.14 . Si hemos definido dicha constante como `const double PI = 3.14` y posteriormente necesitamos cambiar el valor de la misma a 3.1416 , sólo tendremos que modificar una línea, la que define la constante. En cambio, si no hemos declarado `PI`, sino que hemos utilizado el valor 3.14 directamente N veces, tendríamos que realizar N cambios.

DECLARACIÓN DE UNA VARIABLE

Una variable representa un espacio de memoria para almacenar un valor de un determinado tipo. El valor de una variable, a diferencia de una constante, puede cambiar durante la ejecución de un programa. Para utilizar una variable en un programa, primero hay que declararla. La declaración de una variable consiste en enunciar el nombre de la misma y asociarle un tipo:

tipo identificador[, identificador]...

En el ejemplo siguiente se declaran e inician cuatro variables: una de tipo **char**, otra **int**, otra **float** y otra **double**:

```
char c = '\n';
int main()
{
    int i = 0;
    float f = 0.0F;
    double d = 0.0;

    // ...
}
```

El tipo, primitivo o derivado, determina los valores que puede tomar la variable así como las operaciones que con ella pueden realizarse. Los operadores serán expuestos un poco más adelante.

En el ejemplo anterior puede observar que hay dos lugares donde se puede realizar la declaración de una variable: fuera de todo bloque, entendiendo por bloque un conjunto de sentencias encerradas entre el carácter '{' y el carácter '}', y dentro de un bloque de sentencias.

Cuando la declaración de una variable tiene lugar dentro de un bloque, dicha declaración en C++ puede realizarse en cualquier parte, pero siempre antes de ser

utilizada. Se aconseja declarar las variables justo en los lugares donde vayan a ser utilizadas.

En nuestro ejemplo, se ha declarado la variable *c* antes de la función **main** (fuera de todo bloque) y las variables *i*, *f* y *d* dentro de la función (dentro de un bloque). Una variable declarada fuera de todo bloque se dice que es *global* porque es accesible en cualquier parte del código que hay desde su declaración hasta el final del fichero fuente. Por el contrario, una variable declarada dentro de un bloque se dice que es *local* porque sólo es accesible dentro de éste. Para comprender esto mejor, piense que generalmente en un programa habrá más de un bloque de sentencias. No obstante, esto lo veremos con más detalle en el capítulo siguiente.

Según lo expuesto, la variable *c* es global y las variables *i*, *f* y *d* son locales.

Iniciación de una variable

Las variables globales son iniciadas por omisión por el compilador C++: las variables numéricas con *0* y los caracteres con `'\0'`. También pueden ser iniciadas explícitamente, como hemos hecho en el ejemplo anterior con *c*. En cambio, las variables locales no son iniciadas por el compilador C++. Por lo tanto, depende de nosotros iniciarlas o no; es aconsejable iniciarlas, ya que, como usted podrá comprobar, esta forma de proceder evitará errores en más de una ocasión.

OPERADORES

Los operadores son símbolos que indican cómo son manipulados los datos. Se pueden clasificar en los siguientes grupos: aritméticos, relacionales, lógicos, unitarios, a nivel de bits, de asignación, operador condicional y otros. En el capítulo *Introducción a C++* vimos los operadores aritméticos y los de relación de los cuales vemos a continuación algunos ejemplos, para después pasar a describir el resto de operadores.

Operadores aritméticos

El siguiente ejemplo muestra cómo utilizar los operadores aritméticos (+, -, *, / y %). Como ya hemos venido diciendo, observe que primero se declaran las variables y después se realizan las operaciones deseadas con ellas.

```
int a = 10, b = 3, c;  
float x = 2.0F, y;  
y = x + a;           // El resultado es 12.0 de tipo float  
c = a / b;           // El resultado es 3 de tipo int
```

```

c = a % b;           // El resultado es 1 de tipo int
y = a / b;          // El resultado es 3 de tipo int. Se
                    // convierte a float para asignarlo a y
c = x / y;          // El resultado es 0.666667 de tipo float. Se
                    // convierte a int para asignarlo a c (c = 0)

```

Según hemos indicado en el apartado anterior, para realizar la suma $x+a$ el valor del entero a es convertido a **float**, tipo de x . No se modifica a , sino que su valor es convertido a **float** sólo para realizar la suma. Por otra parte, del resultado de x/y sólo la parte entera es asignada a c , ya que c es de tipo **int**. Esto indica que los reales son convertidos a enteros, truncando la parte fraccionaria. No obstante, respecto a esta operación de asignación observaremos que el compilador mostrará un aviso indicando que una conversión de **float** a **int** produce una pérdida de precisión, por si esta apreciación hubiera pasado desapercibida para nosotros. Para evitar que el compilador muestre este aviso tendríamos que especificar explícitamente que deseamos realizar esa conversión así:

```

c = static_cast<int>(x / y); // conversión de x/y a int

```

Un resultado real es redondeado. Observe la operación x/y para x igual a 2 e y igual a 3; el resultado es 0.666667 en lugar de 0.666666 porque la primera cifra decimal suprimida es 6. Cuando la primera cifra decimal suprimida es 5 o mayor de 5, la última cifra decimal conservada se incrementa en una unidad.

Quizás ahora le resulte muy sencillo calcular el área de un determinado triángulo que tenga, por ejemplo, una base de 11,5 y una altura de 3. Veámoslo:

```

#include <iostream>
using namespace std;
int main()
{
    double base = 11.5, altura = 3.0, area = 0.0;

    area = base * altura / 2;
    cout << "Area = " << area << endl;
}

```

Ejecución del programa:

Area = 17.25

Operadores de relación

Recordar que un operador de relación (<, >, <=, >=, != (**not_eq**) y ==) equivale a una pregunta relativa a cómo son dos operandos entre sí. Por ejemplo, la expres-

sión $x=y$ equivale a la pregunta ¿ x es igual a y ? Una respuesta *sí* equivale a un valor **true** y una respuesta *no* equivale a un valor **false**. Por ejemplo:

```
int x = 10, y = 0;
bool r = false;

r = x == y;    // r = false (0) porque x no es igual a y
r = x > y;     // r = true (1) porque x es mayor que y
r = x != y;    // r = true (1) porque x no es igual a y
```

En expresiones largas o confusas, el uso de paréntesis y espacios puede añadir claridad, aunque no sean necesarios. Por ejemplo, las sentencias anteriores serían más fáciles de leer si las escribiéramos así:

```
r = (x == y);    // r = false (0) porque x no es igual a y
r = (x > y);     // r = true (1) porque x es mayor que y
r = (x not_eq y); // r = true (1) porque x no es igual a y
```

Estas sentencias producen los mismos resultados que las anteriores, lo que quiere decir que los paréntesis no son necesarios. ¿Por qué? Porque como veremos un poco más adelante, la prioridad de los operadores `==`, `>` y `!=` es mayor que la del operador `=`, por lo que se evalúan antes que éste. Observe también la utilización de la palabra reservada **not_eq** en lugar del operador `!=`.

Los operadores explícitos como **not_eq**, **and**, **or**, **not**, etc. sólo son soportados por compiladores C++ *estándar*. Estos operadores están definidos en el fichero de cabecera *ciso646*.

Operadores lógicos

El resultado de una operación lógica (AND, OR y NOT) es un valor booleano verdadero o falso (**true** o **false**). Las expresiones que dan como resultado valores booleanos (véanse los operadores de relación) pueden combinarse para formar expresiones *booleanas* utilizando los operadores lógicos indicados a continuación. Los operandos deben ser expresiones que den un resultado **true** o **false**.

En C++, toda expresión numérica con un valor distinto de 0 se corresponde con un valor booleano **true** y toda expresión numérica de valor 0, con **false**.

Operador	Operación
&& o and	Da como resultado verdadero si al evaluar cada uno de los operandos el resultado es verdadero. Si uno de ellos es falso, el resultado es falso. Si el primer operando es falso, el segundo operando no es evaluado.

	u or	El resultado es falso si al evaluar cada uno de los operandos el resultado es falso. Si uno de ellos es verdadero, el resultado es verdadero. Si el primer operando es verdadero, el segundo operando no es evaluado (el carácter es el ASCII 124).
!	o not	El resultado de aplicar este operador es falso si al evaluar su operando el resultado es verdadero, y verdadero en caso contrario.

El resultado de una operación lógica es de tipo **bool**. Por ejemplo:

```
int p = 10, q = 0;
bool r = false;

r = (p != 0) && (q > 0); // r = false (0)
```

En este ejemplo, los operandos del operador `&&` son: $p \neq 0$ y $q > 0$. El resultado de la expresión $p \neq 0$ es verdadero porque p vale 10 y el de $q > 0$ es falso porque q es 0. Por lo tanto, el resultado de *verdadero* `&&` *falso* es falso.

La expresión booleana anterior es equivalente a la siguiente:

```
r = (p not_eq 0) and (q > 0);
```

Los paréntesis que aparecen en la sentencia anterior no son necesarios pero añaden claridad. No son necesarios porque, como veremos un poco más adelante, la prioridad de los operadores de relación es mayor que la de los operadores lógicos, lo que quiere decir que se ejecutan antes.

Operadores unitarios

Los operadores unitarios se aplican a un solo operando y son los siguientes: `!`, `-`, `~`, `++` y `--`. El operador `!` ya lo hemos visto y los operadores `++` y `--` los veremos más adelante.

Operador	Operación
<code>~</code> o compl	Complemento a 1 (cambiar ceros por unos y unos por ceros). El carácter <code>~</code> es el ASCII 126. El operando debe ser de un tipo primitivo entero.
<code>-</code>	Cambia de signo al operando (esto es, se calcula el complemento a 2 que es el complemento a 1 más 1). El operando puede ser de un tipo primitivo entero o real.

El siguiente ejemplo muestra cómo utilizar estos operadores:

```
int a = 2, b = 0, c = 0;
c = -a;           // resultado c = -2
c = compl b;     // resultado c = -1
```

Operadores a nivel de bits

Estos operadores permiten realizar con sus operandos las operaciones AND, OR, XOR y desplazamientos, bit por bit. Los operandos tienen que ser enteros.

Operador	Operación
& o bitand	Operación AND a nivel de bits.
o bitor	Operación OR a nivel de bits (carácter ASCII 124).
^ o bitxor	Operación XOR a nivel de bits.
<<	Desplazamiento a la izquierda rellenando con ceros por la derecha.
>>	Desplazamiento a la derecha rellenando con el bit de signo por la izquierda.

Los operandos tienen que ser de un tipo primitivo entero.

```
int a = 255, r = 0, m = 32;
r = a & 017; // r=15. Pone a cero todos los bits de a
              // excepto los 4 bits de menor peso.
r = r | m;   // r=47. Pone a 1 todos los bits de r que
              // estén a 1 en m.
r = a & ~07; // r=248. Pone a 0 los 3 bits de menor peso de a.
r = a >> 7;  // r=1. Desplazamiento de 7 bits a la derecha.
r = m << 1;  // r=64. Equivale a r = m * 2
r = m >> 1;  // r=16. Equivale a r = m / 2
```

Operadores de asignación

El resultado de una operación de asignación es el valor almacenado en el operando izquierdo, lógicamente después de que la asignación se ha realizado. El valor que se asigna es convertido implícita o explícitamente al tipo del operando de la izquierda (véase el apartado *Conversión entre tipos de datos*). Incluimos aquí los operadores de incremento y decremento porque implícitamente estos operadores realizan una asignación sobre su operando.

Operador	Operación
++	Incremento.
--	Decremento.
=	Asignación simple.

<code>*=</code>	Multiplicación más asignación.
<code>/=</code>	División más asignación.
<code>%=</code>	Módulo más asignación.
<code>+=</code>	Suma más asignación.
<code>-=</code>	Resta más asignación.
<code><<=</code>	Desplazamiento a izquierdas más asignación.
<code>>>=</code>	Desplazamiento a derechas más asignación.
<code>&=</code> o <code>and_eq</code>	Operación AND sobre bits más asignación.
<code> =</code> o <code>or_eq</code>	Operación OR sobre bits más asignación.
<code>^=</code> o <code>xor_eq</code>	Operación XOR sobre bits más asignación.

Los operandos tienen que ser de un tipo primitivo. A continuación se muestran algunos ejemplos con estos operadores.

```
int x = 0, n = 10, i = 1;
n++;           // Incrementa el valor de n en 1.
++n;          // Incrementa el valor de n en 1.
x = ++n;      // Incrementa n en 1 y asigna el resultado a x.
x = n++;      // Equivale a realizar las dos operaciones
               // siguientes en este orden: x = n; n++.
i += 2;       // Realiza la operación i = i + 2.
x *= n - 3;   // Realiza la operación x = x * (n-3) y no
               // x = x * n - 3.
n >>= 1;      // Realiza la operación n = n >> 1 la cual desplaza
               // el contenido de n 1 bit a la derecha.
```

El operador de incremento incrementa su operando en una unidad independientemente de que se utilice como sufijo o como prefijo; esto es, `n++` y `++n` producen el mismo resultado. Ídem para el operador de decremento.

Ahora bien, cuando se asigna a una variable una expresión en la que intervienen operadores de incremento o de decremento, el resultado difiere según se utilicen estos operadores como sufijo o como prefijo. Si se utilizan como prefijo, primero se realizan los incrementos o decrementos y después la asignación (ver más adelante la tabla de prioridad de los operadores). Por ejemplo, `y = ++x` es equivalente a `y = (x += 1)`. En cambio, si se utilizan como sufijo, el valor asignado corresponde a la evaluación de la expresión antes de aplicar los incrementos o los decrementos. Por ejemplo, `y = x++` es equivalente a `y = (t=x, x+=1, t)`, suponiendo que `t` es una variable del mismo tipo que `x`. Esta última expresión utiliza el operador coma que será estudiado un poco más adelante en este mismo capítulo.

Según lo expuesto, ¿cuál es el valor de `x` después de evaluar la siguiente expresión?

```
x = (a - b++) * (--c - d) / 2
```

Comprobemos el resultado evaluando esta expresión mediante el siguiente programa. Observamos que en el cálculo de x intervienen los valores de b sin incrementar y de c decrementado, con lo que el resultado será x igual a 30.

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

int main()
{
    float x = 0, a = 15, b = 5, c = 11, d = 4;

    x = (a - b++) * (--c - d) / 2;
    cout << "x = " << x << ", b = " << b << ", c = " << c << endl;
}
```

Ejecución del programa:

```
x = 30, b = 6, c = 10
```

Una expresión de la complejidad de la anterior equivale a calcular la misma expresión sin operadores $++$ y $--$, pero incrementando/decrementado antes las variables afectadas por $++$ y $--$ como prefijo e incrementado/decrementado después las variables afectadas por $++$ y $--$ como sufijo. Esto equivale a escribir el programa anterior así:

```
int main()
{
    float x = 0, a = 15, b = 5, c = 11, d = 4;

    --c; // o bien c--
    x = (a - b) * (c - d) / 2;
    b++;
    cout << "x = " << x << ", b = " << b << ", c = " << c << endl;
}
```

La aplicación de la regla anterior se complica cuando una misma variable aparece en la expresión, afectada varias veces por los operadores $++$ y $--$ (incluso, reutilizada a la izquierda del signo igual). Por ejemplo:

```
x = (a - b++) * (--b - d) * b++ / (b - d);
```

Cuando se aplica la regla anterior a un caso como éste, hay que tener en cuenta que los incrementos/decrementos como prefijo afectan a los cálculos que le siguen en la propia expresión; por eso habrá que intercalarlos en el lugar adecuado.

En cambio, los incrementos/decrementos como sufijo se aplican igual que antes, al final. El ejemplo siguiente realiza los mismos cálculos que la expresión anterior:

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

int main()
{
    float x = 0, a = 20, b = 10, d = 4;

    x = (a - b);
    --b;
    x *= (b - d) * b / (b - d);
    b++;
    b++;
    cout << "x = " << x << ", b = " << b << endl;
}
```

Ejecución del programa:

x = 90, b = 11

Este código es mucho más sencillo de entender que la expresión equivalente anterior, y también menos propenso a introducir errores, por lo que se recomienda esta forma de trabajar.

Operador condicional

El operador condicional (?), llamado también operador ternario, se utiliza en expresiones condicionales, que tienen la forma siguiente:

$$\text{operando1} ? \text{operando2} : \text{operando3}$$

La expresión *operando1* debe ser una expresión booleana. La ejecución se realiza de la siguiente forma:

- Si el resultado de la evaluación de *operando1* es verdadero, el resultado de la expresión condicional es *operando2*.
- Si el resultado de la evaluación de *operando1* es falso, el resultado de la expresión condicional es *operando3*.

El siguiente ejemplo asigna a *mayor* el resultado de $(a > b) ? a : b$, que será *a* si *a* es mayor que *b* y *b* si *a* no es mayor que *b*.


```
double a = 10.2, b = 20.5, mayor = 0;
mayor = (a > b) ? a : b;
```

Otros operadores

Finalmente vamos a exponer los operadores *global* y de *resolución de ámbito*, el operador *tamaño de*, el operador *coma*, y los operadores *dirección de*, *contenido de* y *referencia a*.

Operador global y de resolución de ámbito (::)

El operador `::` permite acceder a una variable global cuya visibilidad ha sido ocultada por una variable local. El siguiente ejemplo define una variable global `v` y otra local con el mismo nombre, en la función `main`. Observe que para acceder a la variable global se utiliza el operador `::`.

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

float v;

int main()
{
    int v = 7;
    ::v = 10.5; // acceso a la variable global v
    cout << "variable local v = " << v << '\n';
    cout << "variable global v = " << ::v << '\n';
}
```

Después de ejecutar este programa el resultado que se obtiene es el siguiente:

```
variable local v = 7
variable global v = 10.5
```

También se utiliza para especificar a qué clase pertenece un determinado método; por ejemplo, cuando se define fuera del ámbito de la clase, o cuando se invoca si se declaró `static`, cuestión que estudiaremos en el capítulo dedicado a clases de objetos. Por ejemplo, la siguiente línea de código invoca al método `max` declarado `static` en la clase `numeric_limits<float>`:

```
cout << "El float más grande es: " << numeric_limits<float>::max();
```

Operador sizeof

El tamaño de los objetos en C++ es un múltiplo del tamaño de un **char** que por definición es 1 *byte*. Para obtener este tamaño se utiliza el operador **sizeof**. Este operador da como resultado el tamaño en *bytes* de su operando que puede ser el *identificador* o el *tipo* de una variable previamente declarada. Por ejemplo:

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

int main()
{
    int a = 0, t = 0;

    t = sizeof a;
    cout << "El tamaño del entero 'a' es: " << t << " bytes\n"
         << "El tamaño de un entero es: " << sizeof(int) << " bytes\n";
}
```

Ejecución del programa:

```
El tamaño del entero 'a' es: 4 bytes
El tamaño de un entero es: 4 bytes
```

Observe que los paréntesis son opcionales, excepto cuando el operando se corresponde con un tipo de datos. El operador **sizeof** se puede aplicar a cualquier variable de un tipo primitivo o de un tipo derivado, excepto a una matriz de dimensión no especificada, a un campo de bits o a una función.

Operador coma

Un par de expresiones separadas por una coma se ejecutarán de izquierda a derecha y la expresión de la izquierda se descarta. Por ejemplo:

```
int x = 10, y = 0, t = 0;

y = (t=x, x+=1, t); // es equivalente a: y = x++
```

En la sentencia $y = (t=x, x+=1, t)$, se evalúan las tres expresiones especificadas entre paréntesis en el orden en el que están, y se asigna a y el valor que resulte de la última expresión, esto es, de t , las otras dos se descartan.

Otro ejemplo. Supongamos las siguientes llamadas a las funciones $f1$ y $f2$:

```
f1(a++, b-a); // f1 tiene dos parámetros
f2((a++, b-a)); // f2 tiene un sólo parámetro
```

La función *f1* tiene dos parámetros y el orden de evaluación no está definido; dichos parámetros reciben los valores *a* y *b-a*, después *a* se incrementa. En cambio, la función *f2* tiene un solo parámetro (observe los paréntesis internos) que recibe el valor de *b-a*, previamente *a* se incrementa.

Operador dirección-de

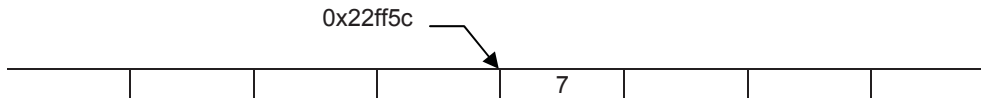
El operador & (dirección de) permite obtener la dirección de su operando. Por ejemplo:

```
int a = 7; // la variable entera 'a' almacena el valor 7
cout << "dirección de memoria = " << &a
    << ", dato = " << a << '\n';
```

El resultado de las sentencias anteriores puede ser similar al siguiente:

```
dirección de memoria = 0x22ff5c, dato = 7
```

El resultado desde el punto de vista gráfico puede verlo en la figura siguiente. La figura representa un segmento de memoria de *n* bytes. En este segmento localizamos el entero 7 de cuatro bytes de longitud en la dirección 0x22ff5c. La variable *a* representa al valor 7 y la expresión &*a* es 0x22ff5c (&*a* - dirección de *a* - es la celda de memoria en la que se localiza *a*).



Este operador no se puede aplicar a un campo de bits perteneciente a una estructura o a un identificador declarado con el calificador **register**, conceptos que veremos más adelante.

Operador de indirección

El operador * (indirección) accede a un valor indirectamente a través de una dirección (un puntero). El resultado es el valor direccionado por el operando; dicho de otra forma, el valor apuntado por el puntero.

Un *puntero* es una variable capaz de contener una dirección de memoria que indica dónde se localiza un dato de un tipo especificado (por ejemplo, un entero). La sintaxis para definir un puntero es:

```
tipo *identificador;
```

donde *tipo* es el tipo del dato apuntado e *identificador* el nombre del puntero (la variable que contiene la dirección de memoria donde está el dato).

El siguiente ejemplo declara un puntero *px* a un valor entero *x* y después asigna este valor al entero *y*.

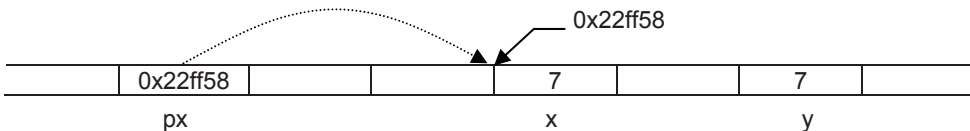
```
#include <iostream>
using namespace std;

int main()
{
    int *px, x = 7, y = 0; // px es un puntero a un valor entero.
    px = &x;              // en el puntero px se almacena la
                          // dirección del entero x.
    y = *px;              // en y se almacena el valor localizado
                          // en la dirección almacenada en px.
    cout << "dirección de memoria = " << &x << ", dato = " << x << '\n';
    cout << "dirección de memoria = " << px << ", dato = " << *px << '\n';
}
```

Ejecución del programa:

```
dirección de memoria = 0x22ff58, dato = 7
dirección de memoria = 0x22ff58, dato = 7
```

Observando el resultado se ve perfectamente que el contenido de *px* (**px*) es 7. La sentencia *y = *px* se lee “y igual al contenido de *px*”. De una forma más explícita diríamos “y igual al contenido de la dirección especificada por *px*”. Gráficamente puede imaginarse esta situación de la forma siguiente:



Observe que una vez que *px* contiene la dirección de *x*, **px* y *x* hacen referencia al mismo dato, por lo tanto, utilizar **px* o *x* es indistinto.

Operador referencia a

Una referencia es un nombre alternativo (un sinónimo) para un objeto. Su utilización la podremos observar en los siguientes capítulos, en el diseño de clases (por ejemplo, en el constructor copia), en el valor retornado por una función para permitir que dicha función sea utilizada a ambos lados del operador de asignación (la función devuelve una referencia), o para permitir que los argumentos en la llama-

da puedan cambiar (paso de parámetros por referencia). La forma general de expresar una referencia es:

$$\text{tipo\& referencia} = \text{variable}$$

El siguiente ejemplo declara una referencia x a una variable y .

```
int y = 10;
int& x = y;
```

Estas sentencias declaran un entero denominado y , e indican al compilador que y tiene otro nombre, x . Las operaciones realizadas sobre y se reflejan en x , y viceversa. Por lo tanto, en operaciones sucesivas, es indiferente utilizar x o y .

Toda referencia, excepto las declaradas como parámetros formales en una función, debe ser siempre iniciada; de lo contrario, el compilador mostrará un error.

Una referencia no es una copia de la variable referenciada, sino que es la misma variable con un nombre diferente. Esto significa, en contra de lo que parece, que un operador no opera sobre la referencia, sino sobre la variable referenciada. Por ejemplo:

```
int conta = 0;
int& con = conta;           // con referencia a conta

con++;                      // conta es incrementado en 1
cout << conta << '\n';     // resultado: 1
cout << con << '\n';       // resultado: 1
```

Obsérvese que aunque $con++$ es correcto, no se incrementa la referencia con , sino que $++$ se aplica al objeto referenciado, que resulta ser el entero identificado por $conta$. Obsérvese también que ambas sentencias **cout** tienen el mismo efecto, puesto que los identificadores $conta$ y con hacen referencia al mismo objeto.

Una referencia, a efectos de resultados, puede ser considerada como un puntero que accede al contenido del objeto apuntado sin necesidad de utilizar el operador de indirección (*). Sin embargo, a diferencia de un puntero, una referencia debe ser iniciada y no puede ser desreferenciada utilizando el operador * (contenido de). Por ejemplo, apoyándonos en el ejemplo anterior, la siguiente línea daría lugar a un error:

```
cout << *con; // indirección ilegal
```

Cuando en una declaración se especifica más de una referencia, cada uno de los identificadores correspondientes debe ser precedido por el operador **&**. Por ejemplo:

```
int m = 10, n = 20;
int& x = m, & y = n, z = n;
```

Este ejemplo define dos referencias, x e y , a m y n , respectivamente, y un entero z , al cual se le ha asignado el valor n .

PRIORIDAD Y ORDEN DE EVALUACIÓN

Cuando escribimos una expresión como la siguiente, $f = a + b * c / d$, es porque conocemos perfectamente el orden en el que se ejecutan las operaciones. Si este orden no fuera el que esperamos tendríamos que utilizar paréntesis para modificarlo, ya que una expresión entre paréntesis siempre se evalúa primero.

Esto quiere decir que el compilador C++ atribuye a cada operador un nivel de prioridad; de esta forma puede resolver qué operación se ejecuta antes que otra en una expresión. Esta prioridad puede ser modificada utilizando paréntesis. Los paréntesis tienen mayor prioridad y son evaluados de más internos a más externos. Como ejemplo de lo expuesto, la expresión anterior puede escribirse también así: $f = (a + ((b * c) / d))$, lo cual indica que primero se evalúa $b * c$, el resultado se divide por d , el resultado se suma con a y finalmente el resultado se asigna a f .

La tabla que se presenta a continuación resume las reglas de prioridad y asociatividad de todos los operadores. Las líneas se han colocado de mayor a menor prioridad. Los operadores escritos sobre una misma línea tienen la misma prioridad.

Operador	Asociatividad
::	ninguna
() [] . -> v++ v-- ..._cast typeid	izquierda a derecha
- + ~ ! * & ++v --v sizeof new delete (tipo)	derecha a izquierda
->* .*	izquierda a derecha
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha

==	!=	izquierda a derecha									
&		izquierda a derecha									
^		izquierda a derecha									
		izquierda a derecha									
&&		izquierda a derecha									
		izquierda a derecha									
?:		derecha a izquierda									
=	*=	/=	%=	+=	-=	<<=	>>=	&=	=	^=	derecha a izquierda
,		izquierda a derecha									

En C++, todos los operadores binarios excepto los de asignación son evaluados de izquierda a derecha. En el siguiente ejemplo, primero se asigna *z* a *y* y a continuación *y* a *x*.

```
int x = 0, y = 0, z = 15;
x = y = z;      // resultado x = y = z = 15
```

CONVERSIÓN ENTRE TIPOS DE DATOS

Anteriormente mencionamos que cuando C++ tiene que evaluar una expresión en la que intervienen operandos de diferentes tipos, primero convierte, sólo para realizar las operaciones solicitadas, los valores de los operandos al tipo del operando cuya precisión sea más alta. Cuando se trate de una asignación, convierte el valor de la derecha al tipo de la variable de la izquierda siempre que no haya pérdida de información; en otro caso, C++ avisará de tal hecho.

Las reglas que se exponen a continuación se aplican en el orden indicado, para cada operación binaria perteneciente a una expresión (dos operandos y un operador), siguiendo el orden de evaluación expuesto en la tabla anterior.

1. Si un operando es de tipo **long double**, el otro operando es convertido a tipo **long double**.
2. Si un operando es de tipo **double**, el otro operando es convertido a tipo **double**.
3. Si un operando es de tipo **float**, el otro operando es convertido a tipo **float**.

4. Un **char** o un **short**, con o sin signo, se convertirán a un **int**, si el tipo **int** puede representar todos los valores del tipo original, o a **unsigned int** en caso contrario.
5. Un **wchar_t** o un tipo enumerado se convierte al primero de los siguientes tipos que pueda representar todos los valores: **int**, **unsigned int**, **long** o **unsigned long**.
6. Un **bool** es convertido en un **int**; **false** se convierte en 0 y **true** en 1.
7. Si un operando es de tipo **unsigned long**, el otro operando es convertido a **unsigned long**.
8. Si un operando es de tipo **long**, el otro operando es convertido a tipo **long**.
9. Si un operando es de tipo **unsigned int**, el otro operando es convertido a tipo **unsigned int**.

Por ejemplo:

```
long a;  
unsigned char b;  
int c;  
float d;  
int f;  
// ...  
f = a + b * c / d;
```

En la expresión anterior se realiza primero la multiplicación, después la división y, por último, la suma. Según esto, el proceso de evaluación será de la forma siguiente:

1. *b* es convertido a **int** (paso 4).
2. *b* y *c* son de tipo **int**. Se ejecuta la multiplicación (*) y se obtiene un resultado de tipo **int**.
3. Como *d* es de tipo **float**, el resultado de *b * c* es convertido a **float** (paso 3). Se ejecuta la división (/) y se obtiene un resultado de tipo **float**.
4. *a* es convertido a **float** (paso 3). Se ejecuta la suma (+) y se obtiene un resultado de tipo **float**.

5. El resultado de $a + b * c / d$, para ser asignado a f , es pasado a entero por truncamiento, esto es, eliminando la parte fraccionaria. Para esta operación el compilador mostrará un aviso indicando que implica una pérdida de precisión.

Cuando el compilador C++ requiere realizar una conversión y no puede, o bien detecta que se incurre en una pérdida de precisión, avisará de tal acontecimiento. En estos casos, lo más normal es resolver tal situación realizando una conversión explícita. No obstante, la conversión explícita de tipos debe evitarse siempre que se pueda porque siempre que se utiliza anula el sistema de chequeo de tipos de C++.

C++ permite una conversión explícita (conversión forzada) del tipo de una expresión mediante una construcción denominada *cast*, que puede expresarse de alguna de las formas siguientes:

- `static_cast<T>(v)`. Este operador convierte la expresión v al tipo T . No examina el objeto que convierte. Se utiliza para realizar conversiones entre tipos relacionados para las que el compilador aplicará una verificación mínima de tipos. Por ejemplo, entre punteros o entre un tipo real y otro entero.
- `reinterpret_cast<T>(v)`. Este operador convierte la expresión v al tipo T . Se utiliza para realizar conversiones entre tipos no relacionados. Se trata de conversiones peligrosas. Por ejemplo, este operador permitiría realizar una conversión de **double** * a **int** * que **static_cast** no permite.
- `dynamic_cast<T>(v)`. Este operador convierte la expresión v al tipo T . Se utiliza para realizar conversiones verificadas durante la ejecución, examinando el tipo del objeto que convierte. Si la conversión es entre punteros y durante la ejecución no se puede realizar, devuelve un cero (puntero nulo).
- `const_cast<T>(v)`. Este operador convierte la expresión v al tipo T . Se utiliza para eliminar la acción ejercida por el calificador **const** sobre v .

Por ejemplo, el siguiente programa escribe la raíz cuadrada de $i/2$ para i igual a 9. Previamente, para obtener un resultado real de la división $i/2$ se convierte el entero i a **double**.

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
    int i = 9;
```

```
double r = 0;

r = sqrt(static_cast<double>(i)/2);
cout << "La raíz cuadrada es " << r << '\n';
}
```

Ejecución del programa:

La raíz cuadrada es 2.12132

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

- 1) ¿Cuál de las siguientes expresiones se corresponde con una secuencia de escape?
 - a) ESC.
 - b) \n.
 - c) \0x07.
 - d) n.

- 2) Los tipos primitivos en C++ son:
 - a) int, float y bool.
 - b) bool, char, short, int, long, float y double.
 - c) char, short, int, long, float, double, long double, wchar_t y bool.
 - d) caracteres, variables y constantes.

- 3) C++ asume que un tipo enumerado es:
 - a) Un tipo entero.
 - b) Un tipo real.
 - c) Un tipo nuevo.
 - d) Una constante.

- 4) 01234 es un literal:
 - a) Decimal.
 - b) Octal.
 - c) Hexadecimal.
 - d) No es un literal.

- 5) 17.24 es un literal de tipo:
- a) char.
 - b) int.
 - c) float.
 - d) double.
- 6) La expresión 's' es:
- a) Una variable.
 - b) Una cadena de caracteres.
 - c) Un entero.
 - d) Un valor de tipo long.
- 7) Se define $a = 5$ y $b = 2$ de tipo **int**. El resultado de a/b es:
- a) 2 de tipo int.
 - b) 2.5 de tipo double.
 - c) 2 de tipo float.
 - d) 2.5 de tipo float.
- 8) Se define $a = 5$ y $b = 2$ de tipo **int**. El valor de $a > b$ es:
- a) 1.
 - b) 2.
 - c) true.
 - d) 0.
- 9) Se define $a = 5$ y $b = 0$ de tipo **int**. El valor de $b = a++$ es:
- a) 4.
 - b) 0.
 - c) 6.
 - d) 5.
- 10) Se define $a = 5$ de tipo **int**. El resultado de $a/static_cast<double>(2)$ es:
- a) 2.
 - b) 2.5.
 - c) 3.
 - d) 0.

2. Escriba un programa que visualice los siguientes mensajes:

Los programas escritos en C++
son portables en código fuente.

- Defina un tipo enumerado *vehículos* con los valores que desee.
- ¿Qué resultados se obtienen al realizar las operaciones siguientes?

```
int a = 10, b = 3, c, d, e;  
float x, y;  
x = a / b;  
c = a < b && 25;  
d = a + b++;  
e = ++a - b;  
y = static_cast<float>(a) / b;
```

- Escriba las sentencias necesarias para visualizar el tamaño en *bytes* de cada uno de los tipos primitivos de C++.
- Escriba un programa que visualice su nombre, dirección y teléfono en líneas diferentes y centrados en la pantalla.
- Escriba un programa que calcule la suma y la media de cuatro valores de tipo **int**.
- Escriba un programa que visualice el resultado de la expresión:

$$\frac{b^2 - 4ac}{2a}$$

para valores de $a = 1$, $b = 5$ y $c = 2$.

ESTRUCTURA DE UN PROGRAMA

En este capítulo estudiará cómo es la estructura de un programa C++. Partiendo de un programa ejemplo sencillo analizaremos cada una de las partes que componen el mismo, así tendrá un modelo para realizar sus propios programas. También veremos cómo se construye un programa a partir de varios módulos fuente. Por último, estudiaremos los conceptos de ámbito y accesibilidad de las variables.

PARADIGMAS DE PROGRAMACIÓN

C++ es un lenguaje que ofrece un abanico tal de utilidades, que lo hacen cómodo para soportar distintos estilos de programación como son la *programación estructurada*, la *programación modular* y la *programación orientada a objetos*.

La unidad básica de la programación estructurada es el procedimiento o función. Desde este estilo de programación un programa C++ se diseña como un conjunto de funciones, las cuales se comunican entre sí mediante el paso de parámetros y la devolución de valores. El concepto de función supone una abstracción que se hace tanto más latente cuanto más grande es el programa.

Un conjunto de procedimientos relacionados, junto con los datos que manipulan, se denomina *módulo*. La aplicación de esta definición condujo a la programación modular, que va mucho más allá de la encapsulación en simples procedimientos; aquí se encapsula un conjunto de operaciones relacionadas que actuarán sobre un conjunto de datos, formando un módulo. El nivel de abstracción está ahora definido por la interfaz que muestra el módulo (funciones del módulo) al usuario para operar sobre el conjunto de datos.

Un módulo define una especie de caja negra que muestra una interfaz al usuario para actuar sobre un conjunto de datos; pero, vista como tal caja negra, no hay

forma de adaptarla a nuevos usos si no es modificando el propio módulo, lo que evidencia poca flexibilidad de adaptación a nuevas necesidades. Este problema tendría solución, en parte, mientras no se alterase el conjunto de datos, si fuera posible construir una jerarquía de módulos en la que cada uno de ellos heredara la funcionalidad de su predecesor; de esta forma añadir nuevas características sería sencillo. Estas ideas, reutilización del código, flexibilidad para añadir nueva funcionalidad, incluso modificar la estructura de datos sobre la que se opera, son las que caracterizan a la programación orientada a objetos que estudiaremos un poco más adelante, a partir del capítulo *Clases*.

C++ fue precisamente diseñado para apoyar la abstracción de datos y la programación orientada a objetos, pero sin forzar a los usuarios a utilizar este estilo de programación, conclusión que usted mismo obtendrá cuando finalice la lectura de este libro.

ESTRUCTURA DE UN PROGRAMA C++

Puesto que C++ es un lenguaje híbrido, vamos a pensar primeramente en un diseño bajo los términos que definen la *programación estructurada* y la *programación modular* y, como ya hemos dicho, más adelante nos centraremos en la *programación orientada a objetos*. Desde esta perspectiva, la solución de cualquier problema no debe considerarse inmediatamente en términos de sentencias correspondientes a un lenguaje, sino de elementos naturales del problema mismo, abstraídos de alguna manera, que darán lugar al desarrollo de las funciones mencionadas y a su agrupación en módulos. Muchas de las funciones que utilizaremos pertenecen a la biblioteca estándar de C++, por lo tanto ya están escritas y compiladas. Pero otras tendremos que escribirlas nosotros mismos, dependiendo del problema que tratemos de resolver en cada caso.

Para explicar cómo es la estructura de un programa C++ desde este punto de vista, vamos a plantear un ejemplo sencillo. Se trata de un programa que muestra una tabla de equivalencia entre grados centígrados y grados *fahrenheit*, según se observa a continuación:

-30 C	-22,00 F
-24 C	-11,20 F
.	.
.	.
90 C	194,00 F
96 C	204,80 F

La relación entre los grados centígrados y los grados *Fahrenheit* viene dada por la expresión $\text{grados Fahrenheit} = 9/5 * \text{grados centígrados} + 32$. Los cálcu-

los los vamos a realizar para un intervalo de -30 a 100 grados centígrados con incrementos de 6 .

Según lo enunciado, vamos a analizar el problema propuesto. ¿Qué piden? Escribir cuántos grados *Fahrenheit* son -30 C, -24 C..., n grados centígrados..., 96 C. Y, ¿cómo hacemos esto? Aplicando la fórmula:

$$\text{GradosFahrenheit} = 9/5 * n\text{GradosCentígrados} + 32$$

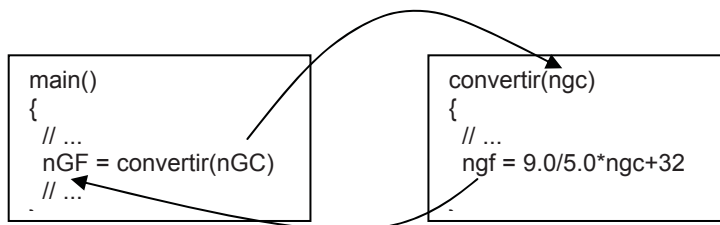
una vez para cada valor de $n\text{GradosCentígrados}$, desde -30 a 100 con incrementos de 6 . Para entender con claridad lo expuesto, hagamos un alto y pensemos en un problema análogo; por ejemplo, cuando nos piden calcular el logaritmo de 2 , en general de n , ¿qué hacemos? Utilizar la función *log*; esto es:

$$x = \log(n)$$

Análogamente, si tuviéramos una función *convertir* que hiciera los cálculos para convertir n grados centígrados en grados *Fahrenheit*, escribiríamos:

$$\text{GradosFahrenheit} = \text{convertir}(n\text{GradosCentígrados})$$

Sin casi darnos cuenta, estamos haciendo una descomposición del problema general en subproblemas más sencillos de resolver. Recordando que un programa C++ tiene que tener una función **main**, además de otras funciones si lo consideramos necesario, ¿cómo se ve de una forma gráfica la sentencia anterior? La figura siguiente da respuesta a esta pregunta:



Análogamente a como hacíamos con la función logaritmo, aquí, desde la función **main**, se llama a la función *convertir* pasándole como argumento el valor en grados centígrados a convertir. La función logaritmo devolvía como resultado el logaritmo del valor pasado. La función *convertir* devuelve el valor en grados *Fahrenheit* correspondiente a los grados centígrados pasados. Sólo queda visualizar este resultado y repetir el proceso para cada uno de los valores descritos. Seguramente pensará que todo este proceso se podría haber hecho utilizando solamente la función **main**, lo cual es cierto. Pero lo que se pretende es que pueda ver de una forma clara que, en general, bajo los términos que definen la *programación es-*

estructurada, un programa C++ es un conjunto de funciones que se llaman entre sí con el fin de obtener el resultado perseguido, y que la forma sencilla de resolver un problema es descomponerlo en subproblemas más pequeños y por lo tanto más fáciles de solucionar; cada subproblema será resuelto por una función C++.

Una vez analizado el problema, vamos a escribir el código. Inicie un nuevo proyecto y añada al fichero *main.cpp* las funciones *convertir* y **main** como se muestra a continuación:

```
/* Paso de grados Centígrados a Fahrenheit (F=9/5*C+32)
 */
#include <iostream>
#include <iomanip>
using namespace std;

// Declaración de la función convertir
float convertir(int c);

int main()
{
    const int INF = -30; // límite inferior del intervalo de °C
    const int SUP = 100; // límite superior */

    // Declaración de variables locales
    int nGradosCentigrados = 0;
    int incremento = 6; // iniciar incremento con 6
    float GradosFahrenheit = 0;

    nGradosCentigrados = INF;
    cout << fixed; // formato en coma flotante
    while (nGradosCentigrados <= SUP)
    {
        // Se llama a la función convertir
        GradosFahrenheit = convertir(nGradosCentigrados);
        // Se escribe la siguiente línea de la tabla
        cout << setw(10) << nGradosCentigrados << " C";
        cout << setw(10) << setprecision(2)
            << GradosFahrenheit << " F\n";
        // Siguiente valor a convertir
        nGradosCentigrados += incremento;
    }
    return 0;
}

// Función convertir grados centígrados a Fahrenheit
float convertir(int gcent)
{
    float gfahr;
```



```
    gfahr = 9.0 / 5.0 * gcent + 32;  
    return (gfahr);  
}
```

Como ya expusimos en un capítulo anterior, siempre que se invoque a una función de la biblioteca de C++, de cualquier otra biblioteca o de nuestros propios recursos, como ocurre con *convertir*, el compilador requiere conocer cómo fue declarada esa función que estamos utilizando, de ahí la declaración anticipada que hemos especificado antes de **main**. Análogamente, los ficheros de cabecera como *iostream* e *iomanip* anticipan las declaraciones de los elementos de la biblioteca de C++ que utilizamos en el código que hemos escrito.

No se preocupe si no entiende todo el código. Ahora lo que importa es que aprenda cómo es la estructura de un programa, no por qué se escriben unas u otras sentencias, cuestión que aprenderá más tarde en éste y en sucesivos capítulos.

A continuación vamos a realizar un estudio de las distintas partes que forman la estructura de un programa. En el ejemplo realizado podemos observar que un programa C++ consta de:

- Directrices **#include** (inclusión de declaraciones y/o definiciones).
- Directrices **using** (definir los espacios de nombres a los que se refiere nuestro programa).
- Función principal **main**.
- Otras funciones y/o declaraciones.

El orden establecido no es esencial, aunque sí bastante habitual. Así mismo, cada función consta de:

- Definiciones y/o declaraciones.
- Sentencias a ejecutar.

Los apartados que se exponen a continuación explican brevemente cada uno de estos componentes que aparecen en la estructura de un programa C++.

Directrices para el preprocesador

La finalidad de las directrices es facilitar el desarrollo, la compilación y el mantenimiento de un programa. Una directriz para el preprocesador se identifica porque empieza con el carácter **#**. Las más usuales son la directriz de inclusión, **#include**, la directriz de definición, **#define**, y las condicionales **#ifndef** y **#endif**. Las directrices son procesadas por el *preprocesador* de C++, que es invocado por el compilador antes de proceder a la traducción del programa fuente.

Inclusión incondicional

En general, cuando se hace uso de un elemento de C++ (constante, variable, clase, objeto, función, etc.), es necesario que dicho elemento esté previamente declarado. Por ejemplo, observe el objeto **cout** en el programa anterior. ¿Cómo sabe el compilador que **cout** admite los argumentos que se han especificado? Pues lo sabe por la información aportada por el fichero de cabecera *iostream*. Lo mismo podríamos decir de las funciones **setw** (ancho) o **setprecision** (precisión); la información respecto a qué parámetros aceptan estas funciones y qué valor retornan es proporcionada por el fichero de cabecera *omanip*. Esto es así para todos los elementos utilizados en el programa, los pertenecientes a la biblioteca de C++ y los definidos por el programador. Las declaraciones de los elementos pertenecientes a la biblioteca de C++ se localizan en los ficheros de cabecera que generalmente se encuentran en el directorio predefinido *include* de la instalación C++.

Según lo expuesto en el apartado anterior, para incluir la declaración de un elemento de la biblioteca de C++ antes de utilizarlo por primera vez, basta con añadir el fichero de cabecera que lo contiene. Esto se hace utilizando la directriz **#include** de alguna de las dos formas siguientes:

```
#include <iostream>
#include "misfuncs.h"
```

Si el fichero de cabecera se delimita por los caracteres `<>`, el preprocesador de C++ buscará ese fichero directamente en las rutas especificadas para los ficheros de cabecera (generalmente esta información la proporciona una variable de entorno del sistema, por ejemplo, *INCLUDE*). En cambio, si el fichero de cabecera se delimita por los caracteres `" "`, el preprocesador de C++ buscará ese fichero primero en el directorio actual de trabajo y si no lo localiza aquí, entonces continúa la búsqueda por las rutas a las que nos hemos referido anteriormente. En cualquier caso, si el fichero no se encuentra se mostrará un error.

Lógicamente, con esta directriz se puede incluir cualquier fichero que contenga código fuente, independientemente de la extensión que tenga.

Definición de un identificador

Mediante la directriz **#define** *identificador* se indica al preprocesador que declare ese *identificador*. Por ejemplo:

```
#define ID
```

Inclusión condicional

Utilizando las directrices **#ifndef** *id* (si no está definido *id*) y **#endif** (fin de si...) se puede especificar que un determinado código sea o no incluido en un programa. La forma de realizar esto es así:

```
#ifndef ID
    #define ID
    // Añadir aquí el código que se desea incluir.
#endif
```

Las directrices especificadas en el código anterior se ejecutan así: si el identificador *ID* aún no ha sido definido, lo cual ocurrirá la primera vez que se ejecute **#ifndef**, esta directriz devolverá un valor distinto de cero (verdadero) y se procesará todo el código que hay hasta la directriz **#endif**, incluida la directriz **#define** que definirá *ID*; si el *ID* ya está definido, lo cual ocurrirá las siguientes veces que se ejecute **#ifndef**, esta directriz devolverá un valor cero (falso) y no se procesará el código que hay hasta la directriz **#endif**. Esta es la técnica que emplea la biblioteca de C++ en sus ficheros de cabecera para que su contenido no sea incluido más de una vez en una unidad de traducción a través de la directriz **#include**, lo que durante la compilación provocaría errores por redefinición.

Definiciones y declaraciones

Una declaración introduce uno o más nombres en un programa. Por lo tanto, todo nombre (identificador) debe ser declarado antes de ser utilizado. ¿Dónde? Se aconseja hacerlo en el mismo lugar donde vaya a ser utilizado por primera vez. La declaración de un nombre implica especificar su tipo para informar al compilador a qué tipo de entidad se refiere. Algunos ejemplos son:

```
class CGrados;           // declaración previa de la clase CGrados
int nGradosCentigrados = 0; // definición de nGradosCentigrados
double sqrt(double);     // declaración de la función sqrt
extern int x;            // declaración de la variable x
float gFahr = 0;        // definición de la variable gFahr
```

Una declaración es una definición, a menos que no haya asignación de memoria como ocurre en las líneas siguientes:

```
double sqrt(double);
extern int x;
class CGrados;
```

La definición de una función (método o procedimiento) declara la función y además incluye el cuerpo de la misma. Por ejemplo:

```
float fx(int z)
{
    // Cuerpo de la función
}
```

La declaración o la definición de una variable puede realizarse a *nivel interno* (dentro del cuerpo de una función) o a *nivel externo* (fuera de toda definición de función), pero la definición de una función no puede realizarse dentro de otra función.

Sentencia simple

Una *sentencia simple* es la unidad ejecutable más pequeña de un programa C++. Las sentencias controlan el flujo u orden de ejecución. Una sentencia C++ puede formarse a partir de: una palabra clave (**for**, **while**, **if ... else**, etc.), expresiones, declaraciones o llamadas a funciones. Cuando se escriba una sentencia hay que tener en cuenta las siguientes consideraciones:

- Toda sentencia simple termina con un punto y coma (;).
- Dos o más sentencias pueden aparecer sobre una misma línea, separadas una de otra por un punto y coma, aunque esta forma de proceder no es aconsejable porque va en contra de la claridad que se necesita cuando se lee el código de un programa.
- Una sentencia nula consta solamente de un punto y coma. Cuando veamos la sentencia **while**, podrá ver su utilización.

Sentencia compuesta o bloque

Una *sentencia compuesta* o bloque es una colección de sentencias simples incluidas entre llaves - { } -. Un bloque puede contener a otros bloques. Un ejemplo de una sentencia de este tipo es el siguiente:

```
{
    GradosFahrenheit = convertir(nGradosCentigrados);
    cout << setw(10) << nGradosCentigrados << " C";
    cout << setw(10) << setprecision(2)
         << GradosFahrenheit << " F\n";
    nGradosCentigrados += incremento;
}
```

Funciones

Una función (unidad de ejecución denominada también procedimiento o método) es un bloque de sentencias que ejecuta una tarea específica y al que nos referimos mediante un nombre. El bloque es el cuerpo de la función y el nombre del bloque es el nombre de la función. Cuando se escribe una función, además del cuerpo y del nombre de la misma, en general hay que especificar también los parámetros en los que se apoyan las operaciones que tiene que realizar y el tipo del resultado que retornará. Por ejemplo:

```

    Tipo del valor          Parámetros que se pasarán como argumentos
    retornado              cuando se invoque a la función
double funcion_a(int p1, float p2)
{
    // declaraciones y sentencias
    return x;
}
    Valor devuelto por
    la función
  
```

Un argumento es el valor que se pasa a una función cuando ésta es invocada. Dicho valor será almacenado en el parámetro correspondiente de la función.

```

int x = 1;
float y = 3.14;
double z = 0;

z = funcion_a(x, y);
    Argumentos pasados a
    la función
  
```

Según lo estudiado hasta ahora, habrá notado que cuando necesitamos realizar un determinado proceso y existe una función de la biblioteca de C++ que lo puede hacer, utilizamos esa función; en otro caso, escribimos nosotros una función que lo realice. Evidentemente se trata de una unidad fundamental en la construcción de módulos y, en definitiva, en la construcción de programas. Por eso, vamos a describir cómo se declaran, cómo se definen y cómo se invocan. Posteriormente, en otros capítulos aprenderá más detalles.

Declaración de una función

La declaración de una función, también conocida como *prototipo de la función*, indica, además del nombre de la función, cuántos parámetros tiene y de qué tipo son, así como el tipo del valor retornado. Su sintaxis es:

```

tipo-resultado nombre-función([lista de parámetros]);
  
```

El prototipo de una función es una plantilla que se utiliza para asegurar que una sentencia de invocación escrita antes de la definición de la función es correcta; esto es, que son pasados los argumentos adecuados para los parámetros especificados en la función y que el valor retornado se trata correctamente. Este chequeo de tipos y número de argumentos permite detectar durante la compilación si se ha cometido algún error.

Por ejemplo, la sentencia siguiente indica que cuando sea invocada la *funcion_a* hay que pasarla dos argumentos, uno entero y otro real, y que dicha función retornará un valor real cuando finalice su ejecución.

```
double funcion_a(int p1, float p2);
```

En conclusión, la declaración de una función permite conocer las características de la misma, pero no define la tarea que realiza.

La función **main** que hemos utilizado hasta ahora siempre devuelve un valor de tipo **int**. Quizás, se pregunte: ¿y por qué en este caso podemos prescindir de la sentencia **return**? Simplemente porque no hace falta, ya que no consta en el programa ninguna llamada a la misma.

La *lista de parámetros* normalmente consiste en una lista de identificadores con sus tipos, separados por comas. En el caso del prototipo de una función, se pueden omitir los identificadores por ser locales a la declaración. Por ejemplo:

```
double funcion_a(int, float);
```

En cambio, cuando se especificuen, su ámbito queda restringido a la propia declaración; esto es, no son accesibles en otra parte (su presencia es sólo para aportar una sintaxis más clara).

De lo expuesto se deduce que los identificadores utilizados en la declaración de la función y los utilizados después en la definición de la misma no necesariamente tienen que nombrarse igual. Observe como ejemplo la declaración y la definición de la *funcion_a* mostradas a continuación:

```
double funcion_a(int x, float y); // declaración de la función

int main()
{
    // ...

    c = funcion_a(a, b); // se llama a la funcion_a
    // ...
}
```

```
double funcion_a(int p1, float p2) // definición de la función
{
    // Cuerpo de la función
}
```

Obsérvese que la *funcion_a* es llamada para su ejecución antes de su definición. Por lo tanto, el nombre de la función, el tipo de los argumentos pasados y el tipo del valor retornado son verificados tomando como referencia la declaración de la función. Si la definición de la función se escribe antes de cualquier llamada a la misma, no es necesario escribir su declaración. Por ejemplo:

```
double funcion_a(int p1, float p2) // definición de la función
{
    // Cuerpo de la función
}

int main()
{
    // ...

    c = funcion_a(a, b); // se llama a la funcion_a
    // ...
}
```

La lista de parámetros puede también estar vacía. Por ejemplo:

```
float funcion_x(); // o bien
float funcion_x(void);
```

Así mismo, para indicar que una función no devuelve nada, se utiliza también la palabra reservada **void**. Por ejemplo:

```
void funcion_x(void)
```

Finalmente, cuando desde una función definida en nuestro programa se invoca a una función de la biblioteca de C++, ¿es necesario añadir su prototipo? Sí es necesario, exactamente igual que para cualquier otra función. Pero no se preocupe, esta tarea resulta sencilla porque las declaraciones de las funciones pertenecientes a la biblioteca estándar de C++, como **sqrt**, son proporcionadas por los ficheros de cabecera. Por eso, cuando un programa utiliza, por ejemplo, la función **sqrt**, observará que se incluye el fichero de cabecera *cmath* (en algunos compiladores basta con incluir *iostream*; a través de éste, se incluyen otros).

```
#include <cmath>
```

¿Cómo conocemos el fichero de cabecera en el que está el prototipo de una determinada función? Porque al especificar la sintaxis de las funciones de la biblioteca de C++, también se indica el fichero de cabecera donde está declarada.

Definición de una función

La definición de una función consta de una *cabecera* de función y del *cuerpo* de la función encerrado entre llaves.

```
tipo-resultado [ámbito::]nombre-función([parámetros formales])
{
    declaraciones de variables locales;
    sentencias;
    [return [(]expresión[)];
}
```

Las variables declaradas en el cuerpo de la función son locales y por definición solamente son accesibles dentro del mismo.

El *tipo del resultado* especifica el tipo de los datos retornados por la función. Éste puede ser cualquier tipo primitivo o derivado, pero no puede ser una matriz o una función. Para indicar que una función no devuelve nada, se utiliza la palabra reservada **void**. Por ejemplo, la siguiente función no acepta argumentos y no devuelve ningún valor:

```
void mensaje(void)
{
    printf("Ocurrió un error al realizar los cálculos\n");
}
```

Cuando una función devuelve un valor, éste es devuelto al punto donde se realizó la llamada a través de la sentencia **return**. La sintaxis de esta sentencia es la siguiente:

```
return [(]expresión[)];
```

La sentencia **return** puede ser o no la última y puede aparecer más de una vez en el cuerpo de la función.

```
int funcion_y(int p1, float p2, char p3)
{
    // ...
    if (a < 0) return 0;
    // ...
    return 1;
}
```


En el ejemplo anterior, si $a < 0$ la función devuelve 0 dando su ejecución por finalizada; si $a \geq 0$ la ejecución continúa y devolverá 1.

Es un error especificar más de un elemento de datos a continuación de **return** (por ejemplo, `return x, y`) ya que el tipo del valor retornado se refiere sólo a uno.

En el caso de que la función no retorne un valor (**void**), se puede especificar simplemente **return** cuando sea necesario, o bien omitir si se trata del final de la función. Por ejemplo:

```
void escribir(void)
{
    // ...
    if (a < 0) return;
    // ...
}
```

La *lista de parámetros formales* de una función está compuesta por las variables que reciben los valores especificados cuando es invocada. Consiste en una lista de cero, uno o más identificadores con sus tipos, separados por comas. El ejemplo siguiente muestra la lista de parámetros de *funcion_x* formada por las variables $p1$, $p2$ y $p3$:

```
float [ámbito::]funcion_x(int p1, float p2, char p3)
{
    // Cuerpo de la función
}
```

Los parámetros formales de una función son variables locales a dicha función. Esto significa que sólo son visibles dentro de la función; dicho de otra forma, sólo tienen vida durante la ejecución de la función.

El *ámbito* es opcional. Indica el ámbito (clase, espacio de nombres, de los que hablaremos un poco más adelante, etc.) al que pertenece la función. Cuando no se especifica, la función pertenece al ámbito global; esto es, se trata de una función global. El siguiente ejemplo incluye una función f perteneciente a un espacio de nombres personalizado (*miesnom*) y otra función f perteneciente al espacio de nombres global. El espacio de nombres sólo contiene la declaración de la función, aunque podría haber contenido su definición. Por eso, al escribir la definición de esta función fuera del cuerpo del espacio de nombres hay que indicar el ámbito al que pertenece, de lo contrario sería tomada como una función global.

```
// Ámbito de una función
#include <iostream>
using namespace std;
```

```
namespace miesnom // espacio de nombres
{
    void f(); // declaración de la función f de miesnom
}

void miesnom::f() // definición de la función f de miesnom
{
    cout << "función f perteneciente al espacio de nombres miesnom\n";
}

void f() // definición de la función f global
{
    cout << "función f perteneciente al ámbito global\n";
}

int main()
{
    f(); // se invoca a la función f global
    miesnom::f(); // se invoca a la función f de miesnom
}
```

Llamada a una función

Lamar o invocar a una función es sinónimo de ejecutarla. La llamada se hará desde otra función o, como veremos más adelante, incluso desde ella misma. Dicha llamada está formada por el nombre de la función seguido de una lista de argumentos, denominados también *parámetros actuales*, encerrados entre paréntesis y separados por comas. Por ejemplo, las siguientes líneas son llamadas a distintas funciones:

```
c = funcion_a(a, b);
f();
miesnom::f();
```

Los argumentos *a* y *b* son las variables, lógicamente definidas previamente, que almacenan los datos que se desean pasar a la *funcion_a*. Una vez finalizada la ejecución de la *funcion_a*, el resultado devuelto por ésta es almacenado en la variable *c* especificada.

Función main

Todo programa C++ tiene una función denominada **main** y sólo una. Esta función es el punto de entrada al programa y también el punto de salida (en condiciones normales de ejecución). Su definición es como se muestra a continuación:

```
int main(int argc, char *argv[])
{
    // Cuerpo de la función
}
```

Como se puede observar, la función **main** está diseñada para devolver un entero (**int**) y tiene dos parámetros que almacenarán los argumentos pasados en la línea de órdenes cuando desde el sistema operativo se invoca al programa para su ejecución, concepto que estudiaremos posteriormente en otro capítulo.

Así mismo, C++ permite escribir **main** sin argumentos:

```
int main()
{
    // Cuerpo de la función
}
```

PASANDO ARGUMENTOS A LAS FUNCIONES

Cuando se llama a una función, el primer argumento en la llamada es pasado al primer parámetro de la función, el segundo argumento al segundo parámetro y así sucesivamente. Por defecto, todos los argumentos, excepto las matrices, son pasados *por valor*. Esto significa que a la función se pasa una copia del valor del argumento. Esto supone que la función invocada trabaje sobre la copia, no pudiendo así alterar las variables de donde proceden los valores pasados.

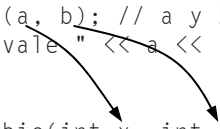
En el siguiente ejemplo, puede observar que la función **main** llama a la función *intercambio* y le pasa los argumentos *a* y *b*. La función *intercambio* almacena en *x* el valor de *a* y en *y* el valor de *b*. Esto significa que los datos *a* y *b* se han duplicado.

```
// Paso de parámetros por valor
#include <iostream>
using namespace std;

void intercambio(int, int); // declaración de la función

int main()
{
    int a = 20, b = 30;
    intercambio(a, b); // a y b son pasados por valor
    cout << "a vale " << a << " y b vale " << b << endl;
}

void intercambio(int x, int y)
{
```



```

int z = x;
x = y;
y = z;
}

```

Veámoslo gráficamente. Supongamos que la figura siguiente representa el segmento de la memoria de nuestro ordenador utilizado por nuestro programa. Cuando se inicia la ejecución de la función **main** se definen las variables *a* y *b* y se inician con los valores 20 y 30, respectivamente. Cuando **main** llama a la función *intercambio*, se definen dos nuevas variables, *x* e *y*, las cuales se inician con los valores de *a* y *b*, respectivamente. El resultado es el siguiente:

	a		b				
	20		30				
					x	y	
					20	30	

Continúa la ejecución de *intercambio*. Se define una nueva variable *z* que se inicia con el valor de *x*. A continuación, en *x* se copia el valor de *y*, y en *y* el valor de *z* (el que tenía *x* al principio). Gráficamente el resultado es el siguiente:

	a		b				
	20		30				
					x	y	
					30	20	
			z				
			20				

Los parámetros *x* e *y* de la función *intercambio* son variables locales a dicha función; lo mismo sucede con *z*. Esto significa que sólo son accesibles dentro de la propia función. Esto se traduce en que las variables locales se crean cuando se ejecuta la función y se destruyen cuando finaliza dicha ejecución. Por lo tanto, una vez que el flujo de ejecución es devuelto a la función **main**, porque *intercambio* finalizó, el estado de la memoria podemos imaginarlo como se observa a continuación, donde se puede observar que *a* y *b* mantienen intactos sus contenidos, independientemente de lo que ocurrió con *x* y con *y*:

	a		b				
	20		30				

Si lo que se desea es alterar los contenidos de los argumentos especificados en la llamada, entonces hay que pasar dichos argumentos *por referencia*. Esto es, a la función hay que pasarla la *dirección* de cada argumento y no su valor, lo que exi-

ge que los parámetros formales correspondientes sean punteros. Para pasar la dirección de un argumento utilizaremos el operador `&`.

Aclaremos esto apoyándonos en el ejemplo anterior. Si lo que queremos es que el intercambio de datos realizado por la función *intercambio* suceda sobre las variables *a* y *b* de **main**, la función *intercambio* lo que tiene que conocer es la posición física que ocupan *a* y *b* en la memoria; de esta forma podrá dirigirse a ellas y alterar su valor. Esa posición física es lo que llamamos dirección de memoria.

Recuerde, en el capítulo 2 se expuso que para que una variable pueda contener una dirección (una dirección es un valor ordinal) hay que definirla así: *tipo *var*. Esta variable recibe el nombre de puntero (apunta al dato) porque su contenido es la posición en la memoria de un determinado dato, no el dato.

Atendiendo a lo expuesto, modifiquemos el ejemplo anterior como se muestra a continuación:

```
// Paso de parámetros por referencia
#include <iostream>
using namespace std;

void intercambio(int *, int *); // declaración de la función

int main()
{
    int a = 20, b = 30;
    intercambio(&a, &b); // a y b son pasados por referencia
    cout << "a es " << a << " y b es " << b << endl;
}

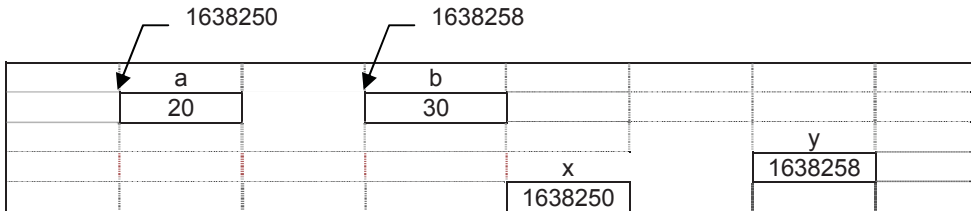
void intercambio(int *x, int *y)
{
    int z = *x; // z = contenido de la dirección x
    *x = *y;    // contenido de x = contenido de y
    *y = z;    // contenido de y = z
}
```

En el ejemplo expuesto podemos ver que la función *intercambio* tiene dos parámetros *x* e *y* de tipo *puntero a un entero* (**int ***), que reciben las direcciones de *a* y de *b*, respectivamente (`&a` y `&b`). Esto quiere decir que cuando modifiquemos el contenido de las direcciones *x* e *y* (`*x` y `*y`), indirectamente estamos modificando los valores *a* y *b*.

Cuando ejecutemos el programa, la función **main** definirá las variables *a* y *b* y llamará a la función *intercambio* pasando las direcciones de dichas variables como argumento. El valor del primer argumento será pasado al primer parámetro y

el valor del segundo argumento, al segundo parámetro. Suponiendo que esas direcciones son 1638250 y 1638258, respectivamente, lo que ocurre es lo siguiente:

```
x = &a; // x = 1638250
y = &b; // y = 1638258
```

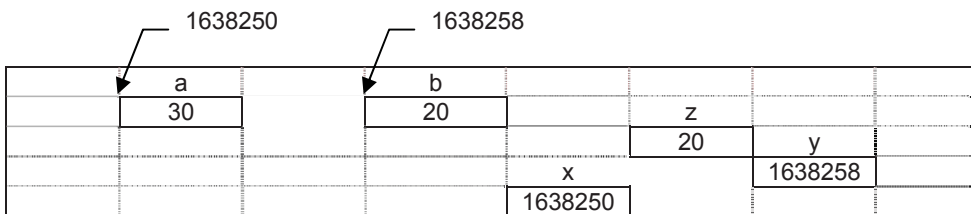


Ahora x apunta al dato a ; esto es, el valor de x especifica el lugar donde se localiza a en la memoria. Análogamente, diremos que y apunta a b .

Observe que $*x$ (contenido de la dirección 1638250; esto es, lo que hay en la casilla situada en esta posición) hace referencia al mismo dato que a y que $*y$ hace referencia al mismo dato que b . Dicho de otra forma $*x$ y a representan el mismo dato, 20. Análogamente, $*y$ y b también representan el mismo dato, 30.

Según lo expuesto, cuando se ejecuten las sentencias de la función *intercambio* indicadas a continuación, el estado de la memoria se modificará como se indica en el gráfico mostrado a continuación:

```
int z = *x; /* z = contenido de la dirección x */
*x = *y; /* contenido de x = contenido de y */
*y = z; /* contenido de y = z */
```



Cuando la función *intercambio* finaliza, los valores de a y b han sido intercambiados y las variables z , x y y , por el hecho de ser locales, son destruidas.

Un ejemplo de la vida ordinaria que explique esto puede ser el siguiente. Un individuo A realiza un programa en C++. Posteriormente envía por correo electrónico a otro individuo B una copia para su corrección. Evidentemente, las correcciones que el individuo B realice sobre la copia recibida no alterarán la que tiene A. Esto es lo que sucede cuando se pasan parámetros por valor.

Ahora, si en lugar de enviar una copia, A le dice a B que se conecte a su máquina 190.125.12.78 y que corrija el programa que tiene almacenado en tal o cual carpeta, ambos, A y B, están trabajando sobre una única copia del programa. Esto es lo que sucede cuando se pasan los argumentos por referencia.

Otra forma de resolver el problema anterior es utilizando referencias en lugar de punteros. Recordemos que una referencia es un nombre alternativo (un sinónimo) para una variable.

Atendiendo a lo expuesto, modifiquemos el ejemplo anterior como se muestra a continuación:

```
// Paso de parámetros por referencia
#include <iostream>
using namespace std;
void intercambio(int&, int&); // declaración de la función

int main()
{
    int a = 20, b = 30;
    intercambio(a, b); // a y b son pasados por referencia
    cout << "a es " << a << " y b es " << b << endl;
}

void intercambio(int& x, int& y)
{
    int z = x; // z = a
    x = y;     // a = b
    y = z;     // b = z
}
```

En el ejemplo expuesto podemos ver que la función *intercambio* tiene dos parámetros *x* e *y* que son referencias, esto es, *x* e *y* son sinónimos de *a* y de *b*, respectivamente. Esto quiere decir que cuando modifiquemos *x* e *y*, estamos modificando los valores *a* y *b*. Esta forma de proceder es más sencilla, sin embargo presenta un inconveniente: observando solamente la llamada a la función *intercambio*, no se sabe si los argumentos son pasados por valor o por referencia.

Resumiendo, pasar parámetros por referencia a una función es hacer que la función acceda indirectamente a las variables pasadas, y a diferencia de cuando se pasan los parámetros por valor, no hay duplicidad de datos.

Cuando se trate de funciones de la biblioteca de C++, también se le presentarán ambos casos.

PROGRAMA C++ FORMADO POR VARIOS MÓDULOS

Según lo que hemos estudiado hasta ahora, no debemos pensar que todo programa tiene que estar escrito en un único fichero *.cpp*. De hecho en la práctica no es así, ya que además del fichero *.cpp*, intervienen uno o más ficheros de cabecera. Por ejemplo, en el programa inicial (conversión de grados) está claro que intervienen los ficheros *main.cpp* e *iostream*, pero, ¿dónde está el código de los elementos de la biblioteca de C++ invocados? Por ejemplo, el programa definido por *main* utiliza el objeto **cout** e invoca a funciones de la biblioteca de C++ como **setw** y **setprecision**, que no están en el fichero *main.cpp*. Estos elementos están definidos en otro fichero separado que forma parte de la biblioteca de C++, al que se accede durante el proceso de enlace para obtener el código correspondiente. Pues bien, nosotros podemos proceder de forma análoga; esto es, podemos optar por escribir el código que nos interese en uno o más ficheros separados (llamados también módulos o unidades de traducción) y utilizar para las declaraciones y/o definiciones uno o más ficheros de cabecera.

Un fichero fuente puede contener cualquier combinación de directrices para el compilador, declaraciones y definiciones. Pero, una función o una clase, en general un bloque, no puede ser dividido entre dos ficheros fuente. Por otra parte, un fichero fuente no necesita contener sentencias ejecutables; esto es, un fichero fuente puede estar formado, por ejemplo, solamente por definiciones de variables que son utilizadas desde otros ficheros fuentes.

Como ejemplo de lo expuesto, vamos a escribir el programa de conversión de grados de otra forma. Esto es, el código correspondiente a dicho programa lo escribiremos en tres ficheros:

- El fichero de cabecera *misfunciones.h* contendrá la declaración de la función *convertir*.
- El fichero *misfunciones.cpp* contendrá la definición de la función *convertir*.
- El fichero *main.cpp* contendrá, además de la función **main**, otras declaraciones que sean necesarias.

Según lo expuesto, escriba el código siguiente en un fichero llamado *misfunciones.h*.

```
#ifndef MISFUNCIONES_H_INCLUDED
#define MISFUNCIONES_H_INCLUDED

// Declaración de la función convertir
float convertir(int c);

#endif // MISFUNCIONES_H_INCLUDED
```


A continuación, escriba este otro código en un fichero llamado *misfunciones.cpp*.

```
// Función convertir grados centígrados a Fahrenheit
float convertir(int gcent)
{
    float gfahr;

    gfahr = 9.0 / 5.0 * gcent + 32;
    return (gfahr);
}
```

Y, finalmente, escriba el código siguiente en el fichero *main.cpp*.

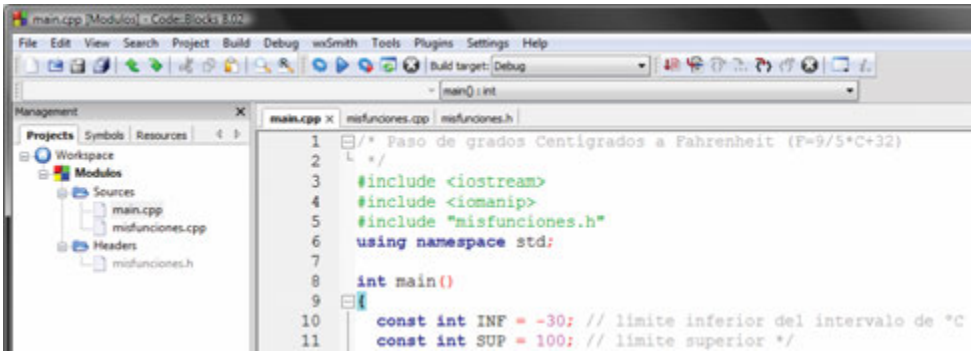
```
/* Paso de grados Centígrados a Fahrenheit (F=9/5*C+32)
 */
#include <iostream>
#include <iomanip>
#include "misfunciones.h"
using namespace std;

int main()
{
    const int INF = -30; // límite inferior del intervalo de °C
    const int SUP = 100; // límite superior */

    // Declaración de variables locales
    int nGradosCentigrados = 0;
    int incremento = 6; // iniciar incremento con 6
    float GradosFahrenheit = 0;

    nGradosCentigrados = INF;
    cout << fixed; // formato en coma flotante
    while (nGradosCentigrados <= SUP)
    {
        // Se llama a la función convertir
        GradosFahrenheit = convertir(nGradosCentigrados);
        // Se escribe la siguiente línea de la tabla
        cout << setw(10) << nGradosCentigrados << " C";
        cout << setw(10) << setprecision(2)
            << GradosFahrenheit << " F\n";
        // Siguiente valor a convertir
        nGradosCentigrados += incremento;
    }
    return 0;
}
```

Observe estos ficheros en la figura siguiente (para añadir un nuevo fichero al proyecto ejecute la orden *File* del menú *File*):

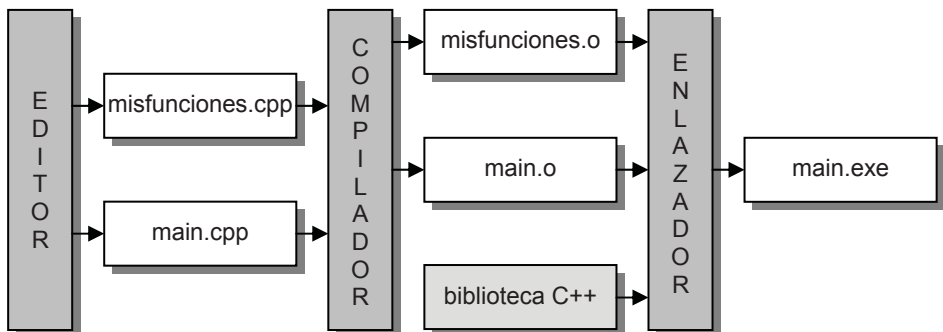


Para compilar un programa formado por varios ficheros, la orden *Build* del menú *Build* del EDI ejecutará una orden análoga a la siguiente:

```
g++ main.cpp misfunciones.cpp -o main.exe
```

En este caso, la orden *g++* junto con la opción *-o*, correspondiente al compilador C++ de GCC (licencia GNU), sería la que tendríamos que escribir en la línea de órdenes para compilar por separado cada uno de los ficheros *.cpp* (los ficheros de cabecera son incluidos por el preprocesador de C++) y, a continuación, enlazarlos para generar el programa ejecutable *main.exe*.

La siguiente figura muestra paso a paso la forma de obtener este fichero. En ella se indica que una vez editados los ficheros *misfunciones.cpp* y *main.cpp*, se compilan obteniéndose como resultado los ficheros objeto *misfunciones.o* y *main.o* (la extensión de estos ficheros depende también de la plataforma sobre la que estemos trabajando), los cuales se enlazan con los elementos necesarios de la biblioteca de C++, obteniéndose finalmente el fichero ejecutable *main.exe*.



ÁMBITO DE UN NOMBRE

Una declaración introduce un nombre (un identificador) en un ámbito, entendiendo por ámbito la parte de un programa donde dicho nombre puede ser referenciado. Un nombre puede ser limitado a un bloque, a un fichero, a una declaración o definición de función, a una clase, a un espacio de nombres, etc.

Nombres globales y locales

Cuando un nombre se declara en un programa fuera de todo bloque (`{ ... }`), es accesible desde su punto de definición o declaración hasta el final del fichero fuente. Este nombre recibe el calificativo de *global*.

Un *nombre global* existe y tiene valor desde el principio hasta el final de la ejecución del programa.

Si la declaración de un nombre se hace dentro de un bloque, el acceso al mismo queda limitado a ese bloque y a los bloques contenidos dentro de éste por debajo de su punto de declaración. En este caso, el nombre recibe el calificativo de *local* o *automático*.

Un *nombre local* existe y tiene valor desde su punto de declaración hasta el final del bloque donde está definido. Cada vez que se ejecuta el bloque que lo contiene, el nombre local es nuevamente definido, y cuando finaliza la ejecución del mismo, el nombre local deja de existir. Por lo tanto, un nombre local es accesible solamente dentro del bloque al que pertenece.

El siguiente ejemplo muestra el ámbito de las variables *var1* y *var2*, dependiendo de si están definidas en un bloque o fuera de todo bloque.

En este ejemplo, analizando el ámbito de las variables, distinguimos cuatro niveles:

- El nivel externo (fuera de todo bloque).

Las variables definidas en este nivel son accesibles desde el punto de definición hasta el final del programa.

- El nivel del bloque de la función **main**.

Las variables definidas en este nivel solamente son accesibles desde la propia función **main** y, por lo tanto, son accesibles en los bloques 1 y 2.

- El nivel del bloque 1 (sentencia compuesta).

Las variables definidas en este nivel solamente son accesibles en el interior del bloque 1 y, por lo tanto, en el bloque 2.

- El nivel del bloque 2 (sentencia compuesta).

Las variables definidas en este nivel solamente son accesibles en el interior del bloque 2.

```
/* Variables globales y locales
*/
#include <iostream>
using namespace std;

// Definición de var1 como variable GLOBAL
int var1 = 50;

int main()
{ // COMIENZO DE main Y DEL PROGRAMA

    cout << var1 << endl; // se escribe 50

    { // COMIENZO DEL BLOQUE 1

        // Definición de var1 y var2 como variables
        // LOCALES en el BLOQUE 1 y en el BLOQUE 2
        int var1 = 100, var2 = 200;

        cout << var1 << " " << var2 << endl; // escribe 100 y 200

        { // COMIENZO DEL BLOQUE 2
            // Redefinición de la variable LOCAL var1
            int var1 = 0;

            cout << var1 << " " << var2 << endl; // escribe 0 y 200

        } // FINAL DEL BLOQUE 2

        cout << var1 << endl; // se escribe 100

    } // FINAL DEL BLOQUE 1

    cout << var1 << endl; // se escribe 50

} // FINAL DE main Y DEL PROGRAMA
```

En el ejemplo anterior se observa que una variable *global* y otra *local* pueden tener el mismo nombre, pero no guardan relación una con otra, lo cual da lugar a que la variable *global* quede ocultada (anulada) en el ámbito de la *local* del mis-

mo nombre. Como ejemplo observe lo que ocurre en el programa anterior con *var1*. No obstante, una variable *global* oculta se puede referenciar utilizando el operador de resolución de ámbito `::`. Por ejemplo, modifique el programa anterior como se indica a continuación y observe los resultados.

```
// FINAL DEL BLOQUE 2

cout << var1 << endl; // se escribe 100
cout << ::var1 << endl; // se escribe 50
```

No hay ninguna forma de utilizar una variable *local* oculta.

CLASES DE ALMACENAMIENTO DE UNA VARIABLE

Por defecto, todas las variables llevan asociada una clase de almacenamiento que determina su accesibilidad y existencia. Los conceptos de accesibilidad y de existencia de las variables pueden alterarse por los calificadores:

auto	almacenamiento automático
register	almacenamiento en un registro
static	almacenamiento estático
extern	almacenamiento externo

Los calificadores **auto** y **register** pueden ser utilizados solamente con variables locales; el calificador **extern** puede ser utilizado sólo con variables globales; y el calificador **static** puede ser utilizado con variables locales y globales.

Calificación de variables globales

Una variable declarada a nivel global es una *definición* de la variable o una *referencia* a una variable definida en otra parte. Las variables definidas a nivel global son iniciadas a 0 por omisión.

Una variable global puede hacerse accesible antes de su definición (si esto tiene sentido) o en otro fichero fuente, utilizando el calificador **extern**. Esto quiere decir que el calificador **extern** se utiliza para hacer visible una variable global allí donde no lo sea.

El siguiente ejemplo formado por dos ficheros fuente, *uno.cpp* y *dos.cpp*, muestra con claridad lo expuesto. El fichero fuente *uno.cpp* define la variable *var* de tipo **int** y le asigna el valor 5. Así mismo, utiliza la declaración **extern int var** para hacer visible *var* antes de su definición.

```
// uno.cpp
#include <iostream>
using namespace std;

void funcion_1();
void funcion_2();

extern int var; // declaración de var que hace referencia a la
                // variable var definida a continuación

int main()
{
    var++;
    cout << var << endl; // se escribe 6
    funcion_1();
}

int var = 5; // definición de var

void funcion_1()
{
    var++;
    cout << var << endl; // se escribe 7
    funcion_2();
}
```

El fichero fuente *dos.cpp* utiliza la declaración **extern int var** para poder acceder a la variable *var* definida en el fichero fuente *uno.cpp*.

```
// dos.cpp
#include <iostream>
using namespace std;

extern int var; // declaración de var. Referencia a la variable var
                // definida en el fichero uno.cpp

void funcion_2()
{
    var++;
    cout << var << endl; // se escribe 8
}
```

Observe que en el programa anterior, formado por los ficheros fuente *uno.cpp* y *dos.cpp*:

1. Existen tres declaraciones externas de *var*: dos en el fichero *uno.cpp* (una definición es también una declaración) y otra en el fichero *dos.cpp*.
2. La variable *var* se define e inicia a nivel global una sola vez; en otro caso se obtendría un error.

3. La declaración **extern** en el fichero *uno.cpp* permite acceder a la variable *var* antes de su definición. Sin la declaración **extern**, la variable global *var* no sería accesible en la función **main**.
4. La declaración **extern** en el fichero *dos.cpp*, permite acceder a la variable *var* en este fichero.
5. Si la variable *var* no hubiera sido iniciada explícitamente, C++ le asignaría automáticamente el valor 0 por ser global.

Si se utiliza el calificador **static** en la declaración de una variable a nivel global, ésta solamente es accesible dentro de su propio fichero fuente. Esto permite declarar otras variables con el mismo nombre en otros ficheros correspondientes al mismo programa. Como ejemplo, sustituya, en el fichero *dos.cpp* del programa anterior, el calificador **extern** por **static**. Si ahora ejecuta el programa observará que la solución es 6, 7, 1 en lugar de 6, 7, 8, lo que demuestra que el calificador **static** restringe el acceso a la variable al propio fichero fuente.

Calificación de variables locales

Una variable local declarada como **auto** (variable automática) solamente es visible dentro del bloque donde está definida. Este tipo de variables no son iniciadas automáticamente, por lo que hay que iniciarlas explícitamente. Es recomendable iniciarlas siempre.

Una variable local declarada **static** solamente es visible dentro del bloque donde está definida; pero, a diferencia de las automáticas, su existencia es permanente, en lugar de aparecer y desaparecer al iniciar y finalizar la ejecución del bloque que la contiene.

Una variable declarada **static** es iniciada solamente una vez, cuando se ejecuta el código que la define, y no es reiniciada cada vez que se ejecuta el bloque que la contiene, sino que la siguiente ejecución del bloque comienza con el valor que tenía la variable cuando finalizó la ejecución anterior. Si la variable no es iniciada explícitamente, C++ la inicia automáticamente a 0.

Una variable local declarada **register** es una recomendación al compilador para que almacene dicha variable, si es posible, en un registro de la máquina, lo que producirá programas más cortos y más rápidos. El número de registros utilizables para este tipo de variables depende de la máquina. Si no es posible almacenar una variable **register** en un registro, se le da el tratamiento de automática. Este tipo de declaración es válido para variables de tipo **int** y de tipo puntero, debido al tamaño del registro.

Una variable declarada **extern** a nivel local hace referencia a una variable definida con el mismo nombre a nivel global en cualquier parte del programa. La finalidad de **extern** en este caso es hacer accesible una variable global, en un bloque donde no lo es.

El siguiente ejemplo clarifica lo anteriormente expuesto.

```
// Variables locales: clases de almacenamiento
#include <iostream>
using namespace std;

void funcion_1();

int main()
{
    // Declaración de var1 que se supone definida en otro sitio
    // a nivel global
    extern int var1;
    // Variable estática var2: es accesible solamente
    // dentro de main. Su valor inicial es 0.
    static int var2;

    // var3 se corresponderá con un registro si es posible
    register int var3 = 0;

    // var4 es declarada auto, por defecto
    int var4 = 0;

    var1 += 2;

    // Se escriben los valores 7, 0, 0, 0
    cout << var1 << " " << var2 << " " << var3 << " " << var4 << endl;
    funcion_1();
}

int var1 = 5;

void funcion_1()
{
    // Se define la variable local var1
    int var1 = 15;

    // Variable estática var2; accesible sólo en este bloque
    static int var2 = 5;

    var2 += 5;
    // Se escriben los valores 15, 10
    cout << var1 << " " << var2 << endl;
}
```


En este ejemplo, la variable global *var1* se define después de la función **main**. Por eso, para hacerla accesible dentro de **main** se utiliza una declaración **extern**. La variable *var2* declarada **static** en **main** es iniciada, por defecto, a 0. La clase de almacenamiento de la variable *var3* es **register** y la de *var4*, **auto**.

En *funcion_1* se define la variable local *var1*; esta definición oculta a la variable global del mismo nombre haciéndola inaccesible dentro de este bloque. La variable *var2*, declarada **static**, es iniciada a 5. Esta definición no entra en conflicto con la variable *var2* de la función **main**, ya que ambas son locales. A continuación, *var2* es incrementada en 5. Entonces, la próxima vez que sea invocada *funcion_1*, iniciará su ejecución con *var2* igual a 10, puesto que las variables locales **static** conservan el valor adquirido de una ejecución para la siguiente.

ESPACIOS DE NOMBRES

El programa ejemplo que acabamos de presentar al comienzo de este capítulo consta de una función, *convertir*, y de la función **main**. Pero en la realidad, los programas son más grandes y, lógicamente, su código está distribuido en múltiples ficheros, cada uno de los cuales puede ser construido y mantenido por diferentes personas o grupos. Esto quiere decir que los desarrolladores deben tener un cuidado especial para no utilizar los mismos identificadores para sus variables, funciones, clases, etc., lo cual resulta difícil y generalmente termina en un gasto de tiempo especial para solucionar estas colisiones entre nombres.

C++ estándar tiene un mecanismo para prevenir este problema: los espacios de nombres. Un *espacio de nombres* es un concepto muy básico y simple: es un ámbito. Por lo tanto, los ámbitos locales y los globales, de los que hemos hablado anteriormente, son espacios de nombres. No obstante, aunque los nombres puedan pertenecer a funciones o a clases, los nombres de funciones globales, los nombres de clases o los nombres de variables globales pertenecen todavía a un único espacio de nombres global. En un proyecto grande, la falta de control sobre estos nombres puede causar problemas. Para adelantarnos a estos problemas, es posible subdividir el espacio de nombres global en espacios personalizados utilizando la palabra reservada **namespace**.

La creación de un espacio de nombres es muy sencilla:

```
namespace nombre_del_espacio
{
    // Declaraciones
}
```

Esto produce un espacio de nombres que empaqueta las declaraciones especificadas en el bloque que define. No es necesario poner un punto y coma después de la llave de cierre.

Los espacios de nombres pueden aparecer solamente en un ámbito global. Por ejemplo:

```
#include <iostream>

namespace LibXXX
{
    int x;
    void f1();
    int f2(bool b);
}

int main()
{
    // ...
}
```

Un espacio de nombres puede estar definido en varios ficheros de cabecera. Por ejemplo:

```
// Espacio de nombres definido en fcab01.h
namespace LibXXX
{
    int x;
    void f1();
    int f2(bool b);
}

// Añadir más nombres a LibXXX en fcab02.h
namespace LibXXX
{
    extern int x;
    int y;
    void f3();
}
```

También es posible declarar un alias de un espacio de nombres:

```
namespace miespnom = LibXXX;
```

En las bibliotecas de clases, los espacios de nombres se utilizan también con frecuencia para empaquetar clases relacionadas entre sí de alguna forma, o bien simplemente para incluir a otros espacios de nombres. Por ejemplo, el código siguiente crearía el espacio de nombres **System** que empaqueta las clases A y B y el

espacio de nombres **Windows**, que a su vez empaqueta el espacio **Forms** que incluye las clases C y D:

```
namespace System
{
    class A {};
    class B {};
    namespace Windows
    {
        namespace Forms
        {
            class C {};
            class D {};
        }
    }
}
```

Para referirnos a un elemento de un espacio de nombres, tenemos que hacerlo utilizando su nombre completo, excepto cuando el espacio de nombres haya sido declarado explícitamente, como veremos a continuación. Por ejemplo, *System::Windows::Forms::C* hace referencia a la clase C del espacio de nombres *System::Windows::Forms*.

```
// ...

int main()
{
    System::Windows::Forms::C obj;
    // ...

    std::cout << LibXXX::x << std::endl;
    // ...
}
```

Toda la biblioteca estándar de C++ está definida en un único espacio de nombres llamado **std** (estándar).

Directriz using

Un programa puede hacer referencia a un nombre de un espacio de nombres de dos formas:

1. Utilizando como prefijo el nombre del espacio en todas las partes del código donde haya que referirse a él. Así, para referirnos a **cout** y **endl** de **std** y a *x* de *LibXXX* escribiríamos:

```
std::cout << LibXXX::x << std::endl;
```

- Indicando al compilador el espacio de nombres donde está el nombre referenciado, lo que posibilita referirse a él sin el nombre de su espacio. Para ello utilizaremos la directriz **using**. Por ejemplo:

```
using namespace std;    // usar el espacio de nombres std

int main()
{
    using namespace LibXXX; // usar el espacio de nombres LibXXX
    // ...
    cout << x << endl;
    // ...
}
```

Como se puede comprobar en el ejemplo anterior, declarar un espacio de nombres permite al programa referirse a sus nombres más tarde sin utilizar el nombre del espacio. Esto es, la directriz **using** sólo indica al compilador C++ dónde encontrar los nombres, no trae nada dentro del programa C++ actual.

En el caso concreto del ejemplo expuesto, si eliminamos la directriz *using namespace std*, el compilador mostrará dos errores para indicar que no puede encontrar los nombres **cout** y **endl**.

La expresión *using namespace miespacio* hace referencia a todos los elementos del espacio de nombres *miespacio*. Para permitir el uso de un elemento determinado, por ejemplo de la variable *x* de *LibXXX*, utilizaríamos una declaración **using** así:

```
void f1()
{
    int x;
    using LibXXX::x; // error: x ya está declarado
    // ...
}
```

Una declaración **using** añade un nombre a un ámbito local; una directriz **using** no, simplemente hace accesibles los nombres de un espacio de nombres.

La directriz **using** puede utilizarse también dentro de un bloque (incluso en el de un espacio de nombres) para hacer disponibles dentro de éste todos los nombres de un espacio de nombres. Esta práctica se recomienda frente al uso de las directrices **using** globales, dejando el uso de éstas últimas para casos excepcionales como, por ejemplo, para la migración de código ya escrito.

EJERCICIOS RESUELTOS

Escriba un programa que utilice:

1. Una función llamada *par_impar* con un parámetro de tipo **int**, que visualice “par” o “impar” respecto del valor pasado como argumento.
2. Una función llamada *positivo_negativo* con un parámetro de tipo **int**, que visualice “positivo” o “negativo” respecto del valor pasado como argumento.
3. Una función llamada *cuadrado* con un parámetro de tipo **int**, que devuelva el cuadrado del valor pasado como argumento.
4. Una función llamada *cubo* con un parámetro de tipo **int**, que devuelva el cubo del valor pasado como argumento.
5. Una función llamada *contar* sin parámetros, que devuelva el siguiente ordinal al último devuelto; el primer ordinal devuelto será el 1.

La función **main** llamará a cada una de estas funciones para un valor determinado y finalmente, utilizando la función *contar*, realizará una cuenta hasta 3.

```
// funciones.cpp - Cómo es un número. Contar.
#include <iostream>
using namespace std;

void par_impar(int);
void positivo_negativo(int);
int cuadrado(int);
int cubo(int);
int contar(void);

int main()
{
    int n = 10;

    par_impar(n);
    positivo_negativo(n);
    cout << "cuadrado de " << n << " = " << cuadrado(n) << endl;
    cout << "cubo de " << n << " = " << cubo(n) << endl;
    cout << "\nContar hasta tres: ";
    cout << contar() << " ";
    cout << contar() << " ";
    cout << contar() << endl;
}

void par_impar(int n)
{
    cout << n << " es " << ((n % 2 == 0) ? "par" : "impar") << endl;
}
```

```
void positivo_negativo(int n)
{
    cout << n << " es " << ((n >= 0) ? "positivo" : "negativo") << endl;
}

int cuadrado(int n)
{
    return n * n;
}

int cubo(int n)
{
    return n * n * n;
}

int contar(void)
{
    static int n = 1;
    return n++;
}
```

Ejecución del programa:

*10 es par
10 es positivo
cuadrado de 10 = 100
cubo de 10 = 1000*

Contar hasta tres: 1 2 3

Este programa, partiendo de un valor n visualiza, invocando a las funciones correspondientes, si este valor es par o impar, positivo o negativo, su cuadrado, su cubo y finalmente realiza una cuenta hasta 3.

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

- 1) Un programa C++ compuesto por dos o más funciones, ¿por dónde empieza a ejecutarse?
 - a) Por la primera función que aparezca en el programa.
 - b) Por la función **main** sólo si aparece en primer lugar.
 - c) Por la función **main** independientemente del lugar que ocupe en el programa.
 - d) Por la función **main** si existe, si no por la primera función que aparezca en el programa.

2) Qué se entiende por preprocesador:

- a) Un programa de depuración.
- b) Un programa cuya tarea es procesar las directrices de un programa C++.
- c) Una herramienta para editar un programa C++.
- d) Una herramienta para compilar un programa C++.

3)Cuál de las siguientes afirmaciones es correcta:

- a) Las directrices incluidas en un programa finalizan con un punto y coma.
- b) Una sentencia compuesta debe finalizar con un punto y coma.
- c) Las directrices deben colocarse al principio del programa.
- d) Una sentencia simple debe finalizar con un punto y coma.

4) La línea siguiente se trata de:

```
float funcion_x(int, float, char);
```

- a) La definición de *funcion_x*.
- b) La declaración de *funcion_x*.
- c) La definición de *funcion_x*, pero falta el cuerpo de la misma.
- d) La declaración de *funcion_x*, pero faltan los nombres de los parámetros.

5)Cuál es el significado que tiene **void**:

- a) Es un tipo de datos.
- b) Permite definir una variable de la cual no se sabe qué tipo de datos va a contener.
- c) Cuando se utiliza en una función permite especificar que no tiene parámetros y/o que no devuelve un resultado.
- d) Cuando se utiliza en una función especifica cualquier número de parámetros.

6) Dado el siguiente código:

```
#include <iostream>
using namespace std;
// Declaración de sumar

int main()
{
    double a = 10, b = 20, c = 0;
    c = sumar(a, b);
    cout << "suma = " << c << endl;
}

double sumar(double x, double y)
{
```

```
    return x + y;  
}
```

¿Cuál o cuáles de las siguientes declaraciones pueden ser escritas en el lugar indicado por el comentario?

- 1.- `double sumar(double x, double y);`
- 2.- `double sumar(double, double);`
- 3.- `double sumar(double a, b);`
- 4.- `double sumar(double a, double b);`

- a) Todas menos la 3.
- b) Todas.
- c) Todas menos la 1 y 3.
- d) Todas menos la 1, 2 y 3.

7) Dado el siguiente código:

```
#include <iostream>  
using namespace std;  
void poner_a_cero(double);  
int main()  
{  
    double a = 10;  
  
    poner_a_cero(a);  
    cout << a << endl;  
}  
void poner_a_cero(double a)  
{  
    double a = 0;  
    return;  
}
```

Al compilar y ejecutar este programa:

- a) Se visualizará 0.
- b) Se visualizará 10.
- c) Se obtendrá un error: sobra la sentencia **return**.
- d) Se obtendrá un error: redefinición del parámetro formal *a*.

8) Dado el siguiente código:

```
#include <iostream>  
using namespace std;  
void poner_a_cero(double);  
int main()  
{  
    double a = 10;
```



```
    poner_a_cero(a);
    cout << a << endl;
}

void poner_a_cero(double a)
{
    a = 0;
    return;
}
```

Al compilar y ejecutar este programa:

- a) Se visualizará 0.
- b) Se visualizará 10.
- c) Se obtendrá un error: sobra la sentencia **return**.
- d) Se obtendrá un error: redefinición del parámetro formal *a*.

9) Dado el siguiente código:

```
#include <iostream>
using namespace std;
void poner_a_cero(double *);

int main()
{
    double a = 10;

    poner_a_cero(&a);
    cout << a << endl;
}

void poner_a_cero(double *a)
{
    *a = 0;
    return;
}
```

Al compilar y ejecutar este programa:

- a) Se visualizará 0.
- b) Se visualizará 10.
- c) Se obtendrá un error: sobra la sentencia **return**.
- d) Se obtendrá un error: redefinición del parámetro formal *a*.

10) Dado el siguiente código:

```
#include <iostream>
using namespace std;
void funcion_x(void);
```

```
int main()
{
    double a = 10;

    funcion_x();
    funcion_x();
    cout << a << endl;
}

void funcion_x(void)
{
    static double a = 5;
    cout << a << endl;
    a++;
}
```

Al compilar y ejecutar este programa:

- a) Se visualizará 5, 5, 10.
 - b) Se visualizará 5, 6, 10.
 - c) Se visualizará 6, 6, 10.
 - d) Se visualizará 10, 11, 12.
2. Escriba el proyecto “convertir grados” y compruebe cómo se ejecuta.
 3. Modifique en el proyecto “convertir grados” los límites inferior y superior de los grados centígrados, el incremento, y ejecute de nuevo el programa.
 4. Modifique en el proyecto “convertir grados” la expresión

```
9.0F/5.0F * gradosC + 32;
```

y escríbala así:

```
9/5 * gradosC + 32;
```

Explique lo que sucede y por qué sucede.

5. De acuerdo con lo expuesto en el apéndice *Entornos de desarrollo* acerca del depurador, pruebe a ejecutar el programa anterior paso a paso y verifique los valores que van tomando las variables a lo largo de la ejecución.

ENTRADA Y SALIDA ESTÁNDAR

Cuando se ejecuta un programa, se suceden fundamentalmente tres tareas: entrada de los datos, proceso de los mismos y salida o presentación de los resultados.



La tarea de entrada obtiene los datos necesarios para el programa de algún medio externo (por ejemplo, del teclado o de un fichero en disco) y los almacena en la memoria del ordenador para que sean procesados; un medio utilizado frecuentemente es el teclado. Por ejemplo:

```
cin >> a >> b; // leer a y b desde el teclado
```

El proceso de los datos dará lugar a unos resultados que serán almacenados temporalmente en memoria. Por ejemplo:

```
c = a + b; // sumar a y b; el resultado se almacena en c
```

La tarea de salida envía los resultados obtenidos a otro lugar; por ejemplo, los visualiza en el monitor, los escribe por la impresora o los guarda en un fichero en disco. La operación de salida no borra los datos de la memoria ni cambia la forma en la que están almacenados. Simplemente hace una copia de los mismos y la envía al lugar especificado; por ejemplo a la pantalla:

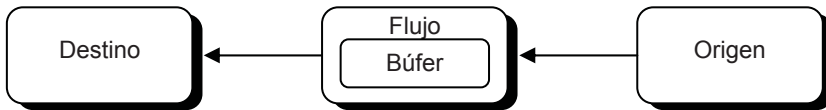
```
cout << c << endl; // mostrar el resultado c en la pantalla
```

En este capítulo estudiaremos los elementos de la biblioteca de C++ que permiten realizar operaciones de entrada y de salida (E/S) sobre los dispositivos

estándar del ordenador; esto es, cómo introducir datos desde el teclado y cómo visualizar datos en la pantalla.

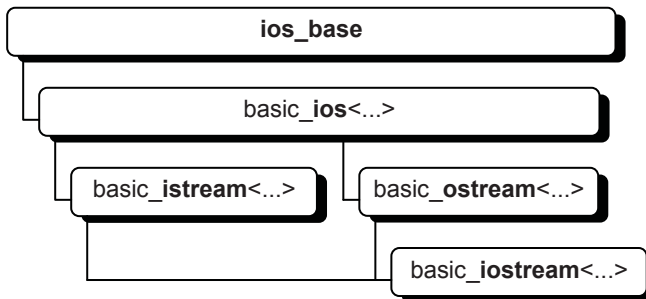
ENTRADA Y SALIDA

Frecuentemente un programa necesitará obtener información desde un origen o enviar información a un destino. Por ejemplo, obtener información desde el teclado, o bien enviar información a la pantalla. La comunicación entre el origen de cierta información y el destino se realiza mediante un *flujo* de información (en inglés *stream*).



Un *flujo* es un objeto que hace de intermediario entre el programa y el origen o el destino de la información. Esto es, el programa leerá o escribirá en el *flujo* sin importarle desde dónde viene la información o a dónde va y tampoco importa el tipo de los datos que se leen o escriben. Este nivel de abstracción hace que el programa no tenga que saber nada ni del dispositivo ni del tipo de información, lo que se traduce en una facilidad más a la hora de escribir programas.

La figura siguiente muestra las clases relacionadas con los flujos vinculados con la entrada y salida estándar. Todas las clases pertenecen al espacio `std`. La clase `ios` permite manipular operaciones generales de E/S. La clase `istream` se deriva de `ios` y permite manipular operaciones de entrada. La clase `ostream` se deriva de `ios` y permite manipular operaciones de salida. Y la clase `iostream` se deriva de `istream` y de `ostream` y permite manipular operaciones de E/S.



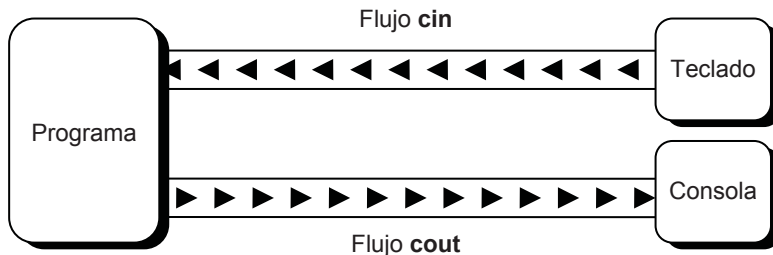
Exactamente, la clase `ios` se obtiene a partir de la plantilla `basic_ios<...>` particularizada para datos de tipo `char` (las plantillas serán estudiadas en un capítulo posterior):

```
typedef basic_ios<char>          ios;
```

Análogamente, las clases **istream**, **ostream** e **iostream** se obtienen, respectivamente, a partir de las plantillas de clase **basic_istream**, **basic_ostream** y **basic_iostream**.

Cuando un programa C++ se ejecuta, se crean automáticamente tres flujos identificados por los objetos indicados a continuación:

- Un flujo desde la entrada estándar (el teclado): **cin**.
- Un flujo hacia la salida estándar (la consola): **cout**.
- Dos flujos hacia la salida estándar de error (la consola): **cerr** y **clog**.



El flujo **cin** es un objeto de la clase **istream** y los flujos **cout**, **cerr** y **clog** son objetos de la clase **ostream**.

Flujos de salida

Cuando un programa define un flujo de salida, por ejemplo el definido por el objeto **cout** de la clase **ostream**, el programa es el origen de ese flujo de bytes (es el que envía los bytes). Esta clase sobrecarga el operador `<<`, que recibe el nombre de operador de inserción, con el fin de ofrecer al programador una notación cómoda que le permita enviar al flujo de salida una secuencia de objetos en una única instrucción (más adelante, en otro capítulo, estudiaremos la sobrecarga de operadores, entendiendo por sobrecarga habilitar a un operador para operar con objetos para los que aún no está habilitado; por ejemplo, se puede habilitar el operador `+` para sumar números complejos). Los objetos **clog** y **cerr** se utilizan de igual manera que **cout**, aunque es habitual reservarlos para informes sobre operaciones realizadas o mensajes de error. Veamos un ejemplo:

```
cout << x; // escribe el valor de x
```

Si x es de tipo **int** con un valor 10, esta sentencia imprimirá 10. Análogamente, si x es de tipo **complex** con un valor (1.5, -2), esta sentencia imprimirá (1.5, -2). Esto sucederá así mientras x sea de un tipo para el que el operador `<<` esté sobrecargado, y lo está para todos los tipos primitivos y algunos derivados como las

cadenas de caracteres y el tipo **string**, y por ahora esto es todo lo que se necesita saber.

Cuando se escriben varias expresiones mediante una única sentencia, éstas se imprimirán en el orden esperado: de izquierda a derecha. Veamos un ejemplo:

```
cout << 'A' << ' ' << static_cast<int>('A') << '\n'; // escribe: A 65
```

Esta sentencia imprime un **char** (A), seguido de otro **char** (espacio en blanco), de un **int** (valor ASCII de A) y de otro **char** (cambio de línea).

El operador de inserción (<<), aunque esté sobrecargado, conserva su precedencia, que es suficientemente baja como para permitir expresiones aritméticas como operandos, sin tener que utilizar paréntesis. Por ejemplo:

```
cout << a+b*c << endl;
```

Esto quiere decir que se deben utilizar paréntesis para escribir expresiones que utilicen operadores con precedencia más baja que <<. Por ejemplo:

```
cout << (a&b|c) << endl;
```

Los argumentos para el operador de inserción pueden ser de cualquier tipo primitivo o derivado predefinido: **string**, **char[]**, **char**, **short**, **int**, **long**, **float**, **double**, **bool**, **unsigned int**, etc. Como ejemplo, el siguiente programa utiliza el operador << para escribir datos de varios tipos en el flujo **cout**.

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

// Tipos de datos
int main()
{
    string sCadena = "Lenguaje C++";
    char cMatrizCars[] = "abc"; // matriz de caracteres
    int dato_int = 4;
    long dato_long = LONG_MIN; // mínimo valor long
    float dato_float = 3.40282347e+38F; // máximo valor float
    double dato_double = 3.1415926; // número PI
    bool dato_bool = true;
    cout << sCadena << endl;
    cout << cMatrizCars << endl;
    cout << dato_int << endl;
    cout << dato_long << endl;
    cout << dato_float << endl;
    cout << dato_double << endl;
```

```

cout << boolalpha; // habilita que se escriba true o false
cout << dato_bool << endl;
}

```

Los resultados que produce el programa anterior son los siguientes:

```

Lenguaje C++
abc
4
-2147483648
3.40282e+038
3.14159
true

```

En el ejemplo se puede observar también que la impresión de un objeto **string** hace que se imprima la cadena de caracteres que almacena.

Flujos de entrada

Cuando un programa define un flujo de entrada, por ejemplo el definido por el objeto **cin** de la clase **istream**, el programa es el destino de ese flujo de bytes (es el que recibe los bytes). Esta clase sobrecarga el operador **>>**, que recibe el nombre de operador de extracción, con el fin de ofrecer al programador una notación cómoda que le permita obtener del flujo de entrada una secuencia de objetos en una única instrucción (sobrecarga: vea el apartado *Flujos de salida*). Por ejemplo:

```
cin >> x; // obtiene el valor de x
```

Si x es de tipo **int** y se teclea un valor 10, esta sentencia asignará 10 a x . Análogamente, si x es de tipo **complex** y se teclea un valor (1.5, -2), esta sentencia asignará (1.5, -2) a x . Esto sucederá así mientras x sea de un tipo para el que el operador **>>** esté sobrecargado, y lo está para todos los tipos primitivos y algunos derivados como las cadenas de caracteres, y por ahora esto es todo lo que se necesita saber.

El siguiente código asigna un valor real a n desde el origen vinculado con el flujo **cin** (entrada estándar):

```

double n = 0;
cout << "Valor = "; // escribe: Valor = (tecleamos 10.5)
cin >> n;           // asigna a n 10.5

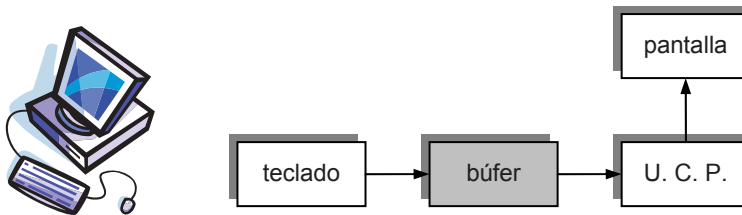
```

Cuando se leen varias variables mediante una única sentencia, a éstas les serán asignados los valores tecleados en el orden esperado: de izquierda a derecha. Por ejemplo:

```
int a;
float b;
char c;
cin >> a >> b >> c;
```

La última sentencia lee un **int**, seguido de un **float** y de un **char**. Estos datos serán introducidos, bien separados uno de otro pulsando la tecla *Entrar*, o bien separados por espacios en blanco, ya que >> salta los espacios en blanco.

Cuando un programa lee datos de la entrada estándar, como hace el operador >> sobre **cin**, la ejecución de dicho programa se detiene hasta que tecleemos los datos que hay que introducir y pulsemos la tecla *Entrar*. Los datos tecleados no son inmediatamente asignados a las variables especificadas y procesados; piense que si esto sucediera así no tendríamos opción a corregir un dato equivocado. Realmente, los datos se escriben en un búfer o memoria intermedia asociada con flujo de entrada y son enviados a la unidad central de proceso cuando se pulsa la tecla *Entrar* para ser asignados a las variables y ser procesados (**cout** y **cerr** también envían los datos al flujo de salida a través de un búfer; **cerr** no utiliza búfer). Esos datos, según se escriben, son visualizados en el monitor con el fin de ver el proceso que estamos realizando.



Cuando se lee un dato, lo que hace el operador de extracción es obtener caracteres del búfer y convertirlos al formato de la variable que va a almacenarlos. En condiciones normales, la asignación a una variable finaliza cuando se llega a un separador. Este proceso se repite para cada una de las variables especificadas.

La ejecución del operador de extracción sobre **cin** finaliza cuando se han asignado valores a todas las variables o cuando se lee un carácter que no se corresponde con el formato de la variable. Por ejemplo, si para la misma sentencia anterior introducimos los datos:

```
5 tm 23.4 z[Entrar]
```

la operación de lectura se interrumpe porque la variable *b* espera un carácter válido para formar un real y 't' no lo es. El resultado es que *a* vale 5 y no se asigna ningún valor ni a *b* ni a *c*. La ejecución continúa en la siguiente sentencia del programa, con los valores que tengan las variables *a*, *b* y *c*. El resultado final será inesperado porque no eran éstos los valores que deseábamos leer.

Los argumentos para el operador de extracción pueden ser de cualquier tipo primitivo o derivado predefinido: **string**, **char[]**, **char**, **short**, **int**, **long**, **float**, **double**, **bool**, **unsigned int**, etc. Como ejemplo, el siguiente programa utiliza el operador `>>` para obtener datos de varios tipos del flujo **cin**.

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

// Tipos de datos
int main()
{
    string sCadena = "Lenguaje C++";
    char cMatrizCars[80]; // matriz de caracteres
    int dato_int = 0;
    long dato_long = 0;
    float dato_float = 0;
    double dato_double = 0;

    cout << "cadena (string): "; cin >> sCadena;
    cout << "cadena (char[]): "; cin >> cMatrizCars;
    cout << "int: "; cin >> dato_int;
    cout << "long: "; cin >> dato_long;
    cout << "float: "; cin >> dato_float;
    cout << "double: "; cin >> dato_double;

    cout << "\ndatos leídos:\n";
    cout << sCadena << endl;
    cout << cMatrizCars << endl;
    cout << dato_int << endl;
    cout << dato_long << endl;
    cout << dato_float << endl;
    cout << dato_double << endl;
}
```

Cuando un flujo se utiliza en una condición, su estado es analizado cada vez que ésta se ejecuta, siendo la condición verdadera solamente si el valor introducido es del tipo de la variable. Veamos un ejemplo:

```
int v = 0;
if (cin >> v)
    cout << v << endl;
else
    cout << "dato incorrecto\n";
```

En este ejemplo, la condición `cin >> v` fallará cuando los caracteres extraídos de la entrada estándar no puedan ser convertidos a un valor de tipo **int**.

ESTADO DE UN FLUJO

La clase **ios_base** es la clase base para todas las clases que definen flujos de E/S. Por lo tanto, sus métodos y atributos serán heredados por sus clases derivadas, lo que permite que un objeto de alguna de estas clases pueda invocar a cualquiera de los métodos heredados públicamente.

Cada flujo tiene un estado asociado con él (dado por un conjunto de bits), que puede ser analizado para manipular cualquier error que pueda ocurrir durante una operación de E/S. Los métodos para examinar el estado de un flujo están definidos en la plantilla **basic_ios** derivada de **ios_base** de la forma siguiente:

```
template< ... >
class basic_ios : public ios_base
{
    // ...

public:
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;
    // ...

};
typedef basic_ios<char>          ios;
```

- ios::good()** Devuelve **true** si todos los bits de error, incluido el de fin de fichero, están a 0. Esto es, la última operación de entrada ha tenido éxito. Por lo tanto, la próxima operación de entrada podría tener éxito; en otro caso, fallará.
- ios::eof()** Devuelve **true** si se encuentra el final de la entrada (*eof*). Esta acción se provoca también cuando por medio del método **clear** se pone el estado del flujo al valor **eofbit**.
- ios::fail()** Devuelve **true** si ocurre cualquier error, excepto *eof*. Esta acción se provoca también cuando por medio del método **clear** se pone el estado del flujo al valor **failbit** o **badbit**, indistintamente. Para saber si este error es recuperable (el flujo está sin corromper y no se han perdido caracteres), compruebe si **bad** devuelve **false**.
- ios::bad()** Devuelve **true** si ocurre un error irrecuperable (el flujo está corrompido). Esta acción se provoca también cuando por medio del método **clear** se pone el estado del flujo al valor **badbit**. En este estado deben abandonarse todas las operaciones de E/S.

Las constantes mencionadas en los párrafos anteriores, utilizadas para modificar el estado de un flujo, están definidas de la forma siguiente:

```
class ios_base
{
public:
    // ...
    static const iostate goodbit,
                          badbit, // error irrecuperable
                          eofbit, // fin de fichero
                          failbit; // error recuperable

    // ...
};
```

ios::goodbit	No hay error (bits de error a 0).
ios::eofbit	Se encontró el final del fichero.
ios::failbit	Posible error recuperable, de formato o de conversión.
ios::badbit	Error irrecuperable.

Por ejemplo, el siguiente código intenta leer un valor **int**. Si el valor introducido para *x* no se corresponde con un entero, se mostrará el mensaje especificado.

```
int x = 0;
cin >> x; // operación de entrada
if (cin.fail())
    cout << "El dato no es correcto\n";
// ...
```

En el código anterior, si la última operación de entrada no tuvo éxito, esto es, si el método **fail** del objeto **cin** devolvió **true**, antes de realizar la próxima operación de entrada hay que poner a 0 todos los indicadores de error, de lo contrario no será posible realizarla, el código se ignorará. Para ello, utilice el método **clear** de la clase **ios** sin argumentos. Este método está definido como se indica a continuación (el valor por omisión del parámetro *f* es **goodbit**; los parámetros con valores por omisión será estudiados en el capítulo *Más sobre funciones*):

```
void clear(iostate f = goodbit);
```

Si *f* es **goodbit**, se desactivan todos los indicadores de error. Si *f* es alguno de los valores **eofbit**, **failbit**, **badbit**, o una combinación de éstos utilizando el operador *or* (**|** o **bitor**), entonces se fija ese estado de error.

DESCARTAR CARACTERES DEL FLUJO DE ENTRADA

Los caracteres que no se extraen del búfer de entrada, por ejemplo, porque ocurrió un error debido a que nos equivocamos al teclearlos, pueden resultar indeseables. Para limpiar este búfer se puede utilizar el método **ignore** de la clase **istream**. Este método está definido así:

```
basic_istream& ignore(streamsize n = 1, int_type delim = traits_type::eof());
```

Este método extrae *n* caracteres (uno por omisión) y los descarta. La ejecución finaliza cuando ocurra algo de lo siguiente:

- Se hayan extraído *n* caracteres, si *n* != **numeric_limits<int>::max()**.
- Se detecte la marca de fin de fichero (*eof*; valor de *delim* por omisión).
- El carácter extraído coincida con el carácter indicado por *delim*.

El método **max** de la plantilla **numeric_limits**, declarada en el fichero de cabecera *limits*, retorna el valor más grande para el tipo especificado.

El siguiente ejemplo implementa una función para leer un dato **float** de forma segura; el dato será solicitado indefinidamente mientras el introducido no sea correcto. Así mismo, este ejemplo muestra cómo utilizar los métodos **clear** e **ignore**, además de otros, cuando ocurre un error recuperable después de un intento de leer datos del flujo de entrada. Obsérvese que, tras un error de entrada, primero, **clear** restablece a **goodbit** el estado del flujo y después, **ignore** elimina los datos que haya en el búfer de entrada hasta encontrar el carácter nueva línea introducido cuando se pulsó la tecla *Entrar*. Escriba la definición de la función en *utils.cpp* y su declaración en *utils.h* para permitir su reutilización en otros programas.

```
#include <iostream>
#include <limits> // necesario para numeric_limits<...>
using namespace std;

float LeerFloat()
{
    float x;
    while(true)
    {
        cin >> x;          // operación de entrada
        if (cin.good())
        {
            cin.ignore(); // eliminar \n del búfer (un carácter)
            return x;     // retornar el valor leído
        }
        else if (cin.fail()) // si la entrada no fue correcta...
        {
```

```

        cin.clear(); // poner a 0 todos los indicadores de error
        cin.ignore(numeric_limits<int>::max(), '\n');
    }
}
}

```

La función anterior puede utilizarse así. Cree un nuevo proyecto y, además de *main.cpp*, añada al mismo *utils.cpp* (véase *Programa c++ formado por varios módulos en el capítulo Estructura de un programa*):

```

#include <iostream>
#include "utils.h" // contiene la declaración de LeerFloat
using namespace std;

int main()
{
    float dato_float = 0;

    cout << "float:          "; dato_float = LeerFloat();
    cout << dato_float << endl;
}

```

ENTRADA/SALIDA CON FORMATO

Los ejemplos realizados hasta ahora han mostrado sus resultados sin formato ninguno. Es decir, lo que hace el operador `<<` sobre el objeto **cout** es convertir los datos a imprimir en una secuencia de caracteres y mostrarlos sin más. Pero es evidente que en muchas ocasiones los resultados hay que mostrarlos según un formato y en un espacio determinado.

La plantilla de clase **basic_ios** derivada de la clase **ios_base** proporciona el control sobre los aspectos de cómo se produce la E/S. Para utilizar esta funcionalidad en una aplicación basta con incluir el fichero de cabecera *iostream*, a través del cual se incluye tanto la funcionalidad proporcionada por esta plantilla de clase como la de **basic_istream** y **basic_ostream** de donde se deriva. Por ejemplo:

```

int v = 165;

cout.setf(ios::hex, ios::basefield); // base 16 (hex)
cout.width(10); // campo de impresión de ancho 10
cout << v << endl; // escribe:          a5

```

Este ejemplo muestra el valor de la variable *v* en hexadecimal, indicado por los argumentos del método **setf** (el primer argumento especifica las opciones que se activan y el segundo, que es opcional, las que se desactivan) y ajustado a la derecha (por omisión) en un campo de ancho 10, indicado por el método **width**.

Para establecer u obtener la base, el tipo de alineación o el tipo de representación, **ios_base** define las siguientes constantes:

```
static const fmtflags basefield;    // dec | oct | hex
static const fmtflags adjustfield; // left | right | internal
static const fmtflags floatfield;  // scientific | fixed
```

Por ejemplo, para verificar si el tipo de alineación es a la izquierda, podemos escribir:

```
if ( (cout.flags() & ios::adjustfield) == ios::left ) ...
```

El método **flags** sobre un flujo devuelve un valor correspondiente al conjunto de opciones establecidas en dicho flujo.

No obstante, para facilitar las operaciones de E/S con formato, la biblioteca estándar ofrece un conjunto de indicadores de manipulación del estado de un flujo que cubren perfectamente la funcionalidad a la que nos hemos referido anteriormente. Estos indicadores son conocidos como *manipuladores*, son insertados directamente en la lista de expresiones de E/S y su finalidad es ejecutar determinadas acciones sobre las operaciones de E/S. Por ejemplo:

```
int v = 0;

cin >> oct >> v; // introducir un valor en base 8; por ejemplo 245
cout << hex << setw(10) << v << endl; // escribe:          a5
cout << dec << setw(10) << v << endl; // escribe:          165
```

Este ejemplo solicita el valor de la variable *v* en octal, indicado por el manipulador **oct**, y lo muestra en hexadecimal y en decimal, indicado por los manipuladores **hex** y **dec**, y ajustado a la derecha (por omisión) en un campo de ancho 10, indicado por el manipulador **setw**. Seguramente, este ejemplo le resultará más fácil de entender que el anterior, en el que las operaciones se escribían en sentencias separadas, perdiendo en cierto modo las conexiones lógicas entre ellas.

Obsérvese que se han realizado dos operaciones justo antes de otra operación de salida. Pues bien, existe una gran variedad de operaciones que en ocasiones será interesante realizar justo antes o después de una operación de E/S. Los manipuladores que las permiten se encuentran en el espacio de nombres **std** y los hay sin parámetros (como **hex**) y con ellos (como **setw**). Muchos de ellos, localizados en los ficheros de cabecera *iostream* (incluye *istream* y *ostream*) e *iomanip*, se resumen a continuación:

boolalpha Permitir mostrar los valores de tipo **bool** en formato alfabético; esta operación se desactiva con **noboolalpha**.

showbase	Permitir mostrar las constantes numéricas precedidas por un dígito distinto de 0, por 0 o por 0x, según se especifiquen en base 10, 8 ó 16, respectivamente; esta operación se desactiva con noshowbase .
showpoint	Forzar a que se muestre el punto decimal y los 0 no significativos en valores expresados en coma flotante; se desactiva con noshowpoint .
showpos	Mostrar el + para los valores positivos; esta operación se desactiva con noshowpos .
skipws	Saltar los espacios en blanco en la entrada (por omisión está activado); esta operación se desactiva con noskipws .
uppercase	Mostrar en mayúsculas los caracteres hexadecimales A-F y la E en la notación científica; esta operación se desactiva con nouppercase .
internal	Hacer que los caracteres de relleno se añadan después del signo o del indicador de base y antes del valor.
left	Alineación por la izquierda y relleno por la derecha.
right	Alineación por la derecha y relleno por la izquierda (establecido por omisión).
dec	Representación en decimal (base por omisión).
oct	Representación en octal.
hex	Representación en hexadecimal.
fixed	Activar el formato de coma flotante (dddd.dd).
scientific	Activar el formato en notación científica (d.dddddEdd).
endl	Escribir ‘\n’ y vaciar el búfer del flujo.
ends	Escribir ‘\0’.
flush	Vaciar el búfer del flujo de salida.
ws	Saltar los espacios en blanco que preceden a un dato en la entrada. Mientras que skipws actúa sobre la entrada en general, ws se aplica sólo sobre el siguiente dato a leer, y se utiliza cuando skipws no está activado.
setiosflags(long)	Activar opciones como por ejemplo fixed , left , etc. (equivalente a setf). Se desactivan con resetiosflags (equivalente a unsetf).
setbase(int)	Establecer la base en la que se escribirán los enteros.
setfill(char)	Establecer como carácter de relleno el especificado.
setprecision(int)	Establecer el número de decimales para un valor real. La precisión por defecto es 6.
setw(int)	Establecer la anchura del campo donde se va a escribir un dato.

El siguiente ejemplo clarifica lo más significativo de lo expuesto hasta ahora.

```
#include <iostream>
#include <iomanip>
```

```

using namespace std;

int main()
{
    int a = 12345;
    float b = 54.865F;

    cout << "          1          2" << endl;
    cout << "12345678901234567890" << endl;
    cout << "-----" << endl;
    cout << a << endl; // escribe 12345\n
    cout << '\n' << setw(10) << "abc"
        << setw(10) << "abcdef" << endl;

    cout << left; // se activa el ajuste a la izquierda
    cout << '\n' << setw(10) << "abc"
        << setw(10) << "abcdef" << endl;
    cout << endl; // avanza a la siguiente línea
    cout << right; // se vuelve al ajuste por la derecha

    // Se activa el formato de coma fija
    // con dos decimales
    cout << fixed << setprecision(2);
    cout << setw(15) << b << endl;
    cout << setw(15) << b/10 << endl;

    return 0;
}

```

Al ejecutar este programa se obtendrán los resultados mostrados a continuación. Observe que `\n` o **endl** avanza al principio de la línea siguiente; si en este instante se envía a la salida otro `\n` o **endl**, estos dan lugar a una línea en blanco.

```

          1          2
12345678901234567890
-----
12345
      abc      abcdef
abc          abcdef
          54.87
          5.49

```

A continuación, damos una explicación de cada uno de los formatos empleados.

```
cout << a << endl; // escribe 12345\n
```


- escribe el entero a .
- **endl** avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << '\n' << setw(10) << "abc" << setw(10) << "abcdef" << endl;
```

- $\backslash n$ avanza a la línea siguiente.
- $setw(10)$ para escribir la cadena “ abc ” sobre un ancho de 10 posiciones. La cadena se ajusta por defecto a la derecha.
- $setw(10)$ para escribir la cadena “ $abcdef$ ” sobre un ancho de 10 posiciones. La cadena se ajusta por defecto a la derecha.
- **endl** avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << left;
```

- activa el ajuste por la izquierda en el ancho que se establezca.

```
cout << '\n' << setw(10) << "abc" << setw(10) << "abcdef" << endl;
```

- igual que anteriormente pero con ajustes a la izquierda.

```
cout << endl;
```

- avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << right;
```

- activa el ajuste por la derecha en el ancho que se establezca.

```
cout << fixed << setprecision(2);
```

- activa el formato de coma fija con dos decimales.

```
cout << setw(15) << b << endl;
```

- $setw(15)$ para escribir la cadena b sobre un ancho de 15 posiciones. El ajuste está establecido por la derecha. El valor de b se redondea a dos decimales.
- **endl** avanza a la línea siguiente y vacía el búfer de la salida.

```
cout << setw(15) << b/10 << endl;
```

- análoga a la anterior. Observe cómo las dos últimas cantidades escritas quedan ajustadas por su parte entera y decimal.

Este otro ejemplo que se presenta a continuación ilustra cómo se utilizan los indicadores de formato a través de **setiosflags**. El resultado que se quiere obtener es el siguiente:

```
Madrid.....5198.00
Sevilla.....3.21
```

```
Valencia.....46.32
Cantabria.....506.50
Barcelona.....2002.38
```

Tanto los nombres de las provincias como los coeficientes asociados estarán almacenados en sendas matrices (las matrices las estudiaremos en un capítulo posterior).

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

int main()
{
    double coef[] = { 5198.0, 3.21, 46.32, 506.5, 2002.38 };
    string prov[] = { "Madrid", "Sevilla", "Valencia", "Cantabria",
                     "Barcelona" };
    // Salida de resultados alineados en columnas
    cout << setiosflags( ios::fixed ); // formato en coma flotante
    for ( int i = 0; i < sizeof( coef )/sizeof( double ); i++)
        cout << setiosflags( ios::left ) // justificación a la izda.
             << setw( 15 ) // ancho para las cadenas de caracteres
             << setfill( '.' ) // carácter de relleno
             << prov[i] // escribe la provincia
             << resetiosflags( ios::left ) // suprime justificación
             << setw( 10 ) // ancho para las cantidades
             << setprecision( 2 ) // dos decimales
             << coef[i] << endl; // escribe cantidad y '\n'
}
```

La sentencia **for**, que será estudiada en un capítulo posterior, tiene como misión ejecutar la sentencia de salida tantas veces como elementos hemos almacenado en las matrices.

Se puede especificar como argumento de **setiosflags** varios indicadores de formato unidos por el operador | (*or*).

ENTRADA DE CARACTERES

El operador **>>** sobre **cin** está pensado para aceptar valores separados por espacios para variables de un tipo esperado. Cuando lo que se desea es leer caracteres como tales (incluidos los espacios) se utiliza el método **get** de **basic_istream**. Cada vez que se ejecute este método se leerá el siguiente carácter al último leído. Su sintaxis es así:

```
basic_istream& get(char_type& car);
```

El método **get** almacena en *car* el carácter leído; si el carácter leído coincide con el final del fichero, pone el estado del flujo al valor **eofbit**. Por ejemplo:

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

int main()
{
    char car = 0;
    cout << "Introducir un carácter: ";
    // Leer un carácter y almacenarlo en la variable car
    cin.get(car);
    cout << "Carácter: " << car
         << ", valor ASCII: " << static_cast<int>(car) << endl;
}
```

Ejecución del programa

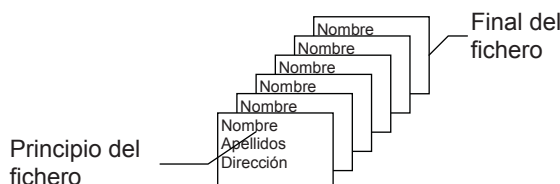
*Introducir un carácter: a
Carácter: a, valor ASCII: 97*

En el ejemplo anterior se puede observar que dependiendo del formato que se utilice para visualizar *car*, **char** o **int**, el resultado puede ser el propio carácter o su valor ASCII. Internamente, refiriéndonos a la memoria del ordenador, no hay más que un conjunto de 0 y 1 (*01100001*).

Suponiendo que el búfer asociado con la entrada estándar está vacío, cuando en el programa anterior se ejecute el método **get**, su ejecución se detendrá hasta que tecleemos un carácter y pulsemos la tecla *Entrar*. El carácter leído será almacenado en la variable *car*.

CARÁCTER FIN DE FICHERO

Desde el punto de vista del desarrollador de una aplicación, un dispositivo de entrada o de salida estándar es manipulado por el lenguaje C++ como si de un fichero de datos en el disco se tratara. Un fichero de datos no es más que una colección de información. Los datos que introducimos por el teclado son una colección de información y los datos que visualizamos en el monitor son también una colección de información.



Todo fichero tiene un principio y un final. ¿Cómo sabe un programa que está leyendo datos de un fichero, que se ha llegado al final del mismo y, por lo tanto, no hay más datos? Por una marca de fin de fichero. En el caso de un fichero grabado en un disco esa marca estará escrita al final del mismo. En el caso del teclado la información procede de lo que nosotros tecleamos, por lo tanto si nuestro programa requiere detectar la marca de fin de fichero, tendremos que teclearla cuando demos por finalizada la introducción de información. Esto se hace pulsando las teclas *Ctrl+D* en UNIX o *Ctrl+Z* en una aplicación de consola en Windows.

Ya que un fichero o un dispositivo siempre son manejados a través de un flujo, hablar del final del flujo es sinónimo de hablar del final del fichero. Por eso, de ahora en adelante, siempre que tengamos que realizar algún tipo de operación sobre un dispositivo o sobre un fichero, nos referiremos indistintamente a ellos o al flujo que los representa.

Cuando el operador `>>` sobre `cin` intenta leer un carácter fin de fichero, pone el estado del flujo al valor `eofbit` (ídem para `get`). También activa el indicador `failbit` puesto que la entrada no es válida. Utilice `clear` para desactivar los indicadores y permitir realizar nuevas lecturas. Veamos un ejemplo. El programa siguiente escribe la suma de los datos introducidos por el teclado. La entrada finalizará cuando se introduzcan *n* datos o se detecte el *eof* (*Ctrl+Z* | *Ctrl+D* seguido de la tecla *Entrar*).

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

int main()
{
    int dato, suma = 0, n = 10, i = 0;
    bool eof = false;

    cout << "Introducir n datos máximo. Finalizar con eof.\n\n";

    do
    {
        cout << "dato: ";
        cin >> dato;
        eof = cin.eof();
        if (!eof && cin.fail())
        {
            cerr << '\a' << "\ndato incorrecto\n";
            cin.clear();    // desactivar los bits de error
            cin.ignore(numeric_limits<int>::max(), '\n');
        }
        else if (cin.good())
        {
```


ya que el carácter `\n` no es un carácter válido para un valor **float**; por lo tanto, aquí se interrumpe la lectura. Este carácter sobrante puede ocasionarnos problemas si a continuación se ejecuta otra sentencia de entrada que admita datos que sean caracteres como sucede con **get**. Por ejemplo, para ver este detalle vamos a modificar el programa anterior de la siguiente forma:

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

int main()
{
    float precio = 0.0F;
    char car = 0;

    cout << "Precio: "; cin >> precio;
    cout << "Pulse <Entrar> para continuar ";
    cin.get(car);
    cout << "Precio = " << precio << endl;
}
```

Si ejecutamos este programa y tecleamos el dato 1000, se producirá el siguiente resultado:

```
Precio: 1000 [Entrar]
Pulse <Entrar> para continuar Precio = 1000
```

A la vista del resultado, se observa que no se ha hecho una pausa. ¿Por qué? Porque el carácter sobrante nueva línea es un carácter válido para el método **get**, razón por la que **get** no necesita esperar a que introduzcamos un carácter para la variable *car*.

La solución al problema planteado es limpiar el búfer asociado con **cin** antes de ejecutarse **get**, utilizando el método **ignore**.

```
int main()
{
    float precio = 0.0F;
    char car = 0;

    cout << "Precio: "; cin >> precio;
    cout << "Pulse <Entrar> para continuar ";
    cin.ignore(); // eliminar el '\n' sobrante
    cin.get(car);
    cout << "Precio = " << precio << endl;
}
```

ENTRADA DE CADENAS DE CARACTERES

El operador `>>` sobre `cin` está pensado para aceptar valores separados por espacios. Entonces, ¿cómo leemos una cadena de caracteres de tipo `string` que contenga espacios en blanco? Pues invocando a la función `getline`:

```
istream& getline (istream& is, string& str);
```

Esta función lee caracteres de la entrada estándar (representada por `cin`) hasta encontrar el carácter `'\n'` que se introduce al pulsar la tecla *Entrar*, incluido este carácter. El hecho de leer también el carácter `'\n'` puede presentar un problema cuando antes se haya leído otro dato de un tipo primitivo. Por ejemplo:

```
cout << "Introducir un real: ";
cin >> dDato;
cout << "Introducir una cadena de caracteres: ";
getline(cin, strDato);
// ...
```

En este ejemplo, `getline` lee el carácter `'\n'` que se introduce al pulsar la tecla *Entrar* después de introducir el valor real, asumiendo que esta es la cadena que queríamos leer (se trata de una cadena vacía), con lo que el usuario no podrá introducir la cadena solicitada. La solución es eliminar ese carácter `'\n'` sobrante antes de que sea solicitada la entrada de la cadena de caracteres:

```
// ...
cout << "Introducir un real: ";
cin >> dDato;
cin.ignore(); // eliminar el '\n' sobrante
cout << "Introducir una cadena de caracteres: ";
getline(cin, strDato);
// ...
```

EJERCICIOS RESUELTOS

1. Realizar un programa que dé como resultado los intereses producidos y el capital total acumulado de una cantidad c , invertida a un interés r durante t días.

La fórmula utilizada para el cálculo de los intereses es:

$$I = \frac{c * r * t}{360 * 100}$$

siendo:

I = Total de intereses producidos.

c = Capital.

r = Tasa de interés nominal en tanto por ciento.

t = Período de cálculo en días.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double c, intereses, capital;
float r;
int t;
```

- A continuación leemos los datos c , r y t .

```
cout << "Capital invertido          "; cin >> c;
cout << "\nA un % anual del          "; cin >> r;
cout << "\nDurante cuántos días        "; cin >> t;
```

- Conocidos los datos, realizamos los cálculos. Nos piden los intereses producidos y el capital acumulado. Los intereses producidos los obtenemos aplicando directamente la fórmula. El capital acumulado es el capital inicial más los intereses producidos.

```
intereses = c * r * t / (360 * 100);
capital = c + intereses;
```

- Finalmente, escribimos el resultado.

```
cout << fixed << setprecision(2);
cout << "Intereses producidos.." << setw(12) << intereses << endl;
cout << "Capital acumulado....." << setw(12) << capital << endl;
```

El programa completo se muestra a continuación.

```
// Capital e Intereses
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    double c, intereses, capital;
    float r;
    int t;

    // Entrada de datos
```



```

cout << "Capital invertido          "; cin >> c;
cout << "\nA un % anual del          "; cin >> r;
cout << "\nDurante cuántos días      "; cin >> t;
cout << "\n\n";

// Cálculos
intereses = c * r * t / (360L * 100);
capital = c + intereses;

// Escribir resultados
cout << fixed << setprecision(2);
cout << "Intereses producidos.." << setw(12) << intereses << endl;
cout << "Capital acumulado....." << setw(12) << capital << endl;
}

```

Ejecución del programa:

```

Capital invertido          1000000
A un % anual del          8
Durante cuántos días      360

Intereses producidos..    80000.00
Capital acumulado.....  1080000.00

```

2. Realizar un programa que calcule las raíces de la ecuación:

$$ax^2 + bx + c = 0$$

teniendo en cuenta los siguientes casos:

- Si a es igual a 0 y b es igual a 0 , imprimiremos un mensaje diciendo que la ecuación es degenerada.
- Si a es igual a 0 y b no es igual a 0 , existe una raíz única con valor $-c/b$.
- En los demás casos, utilizaremos la fórmula siguiente:

$$x_i = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

La expresión $d = b^2 - 4ac$ se denomina discriminante.

- Si d es mayor o igual que 0 entonces hay dos raíces reales.
- Si d es menor que 0 entonces hay dos raíces complejas de la forma:

$$x + yj, x - yj$$

Indicar con literales apropiados los datos a introducir, así como los resultados obtenidos.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.


```
double a, b, c; // coeficientes de la ecuación
double d;      // discriminante
double re, im; // parte real e imaginaria de la raíz
```

- A continuación leemos los datos a , b y c .

```
cout << "Coeficientes a, b y c de la ecuación: ";
cin >> a >> b >> c;
```

- Léidos los coeficientes, pasamos a calcular las raíces.

```
if (a == 0 && b == 0)
    cout << "La ecuación es degenerada\n";
else if (a == 0)
    cout << "La única raíz es: " << -c / b << endl;
else
{
    // Evaluar la fórmula. Cálculo de d, re e im
    if (d >= 0)
    {
        // Imprimir las raíces reales
    }
    else
    {
        // Imprimir las raíces complejas conjugadas
    }
}
```

- Cálculo de $\frac{-b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}$
- 

```
re = -b / (2 * a);
d = b * b - 4 * a * c;
im = sqrt(fabs(d)) / (2 * a);
```

- Imprimir las raíces reales.

```
cout << "Raíces reales:\n";
```

```
cout << re + im << " y " << re - im << endl;
```

- **Imprimir las raíces complejas conjugadas.**

```
cout << "Raíces complejas:\n";
cout << re << " + " << fabs(im) << " j" << endl;
cout << re << " - " << fabs(im) << " j" << endl;
```

El programa completo se muestra a continuación.

```
// ecu2gra.cpp - Raíces de una ecuación de 2º grado
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
    double a, b, c; // coeficientes de la ecuación
    double d;      // discriminante
    double re, im; // parte real e imaginaria de la raíz

    cout << "Coeficientes a, b y c de la ecuación: ";
    cin >> a >> b >> c;
    cout << '\n';

    cout << fixed << setprecision(2);
    if (a == 0 && b == 0)
        cout << "La ecuación es degenerada\n";
    else if (a == 0)
        cout << "La única raíz es: " << -c / b << endl;
    else
    {
        re = -b / (2 * a);
        d = b * b - 4 * a * c;
        im = sqrt(fabs(d)) / (2 * a);
        if (d >= 0)
        {
            cout << "Raíces reales:\n";
            cout << re + im << " y " << re - im << endl;
        }
        else
        {
            cout << "Raíces complejas:\n";
            cout << re << " + " << fabs(im) << " j" << endl;
            cout << re << " - " << fabs(im) << " j" << endl;
        }
    }
}
```

Ejecución del programa

Coeficientes a, b y c de la ecuación: $1 - 2 + 3$

Raíces complejas:

$1.00 + 1.41 j$

$1.00 - 1.41 j$

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

1) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    float a = 0.0F;
    a = 1/3;
    cout<< fixed << a << endl;
}
```

- a) 0.000000.
- b) 0.333333.
- c) 1.0.
- d) 0.33333.

2) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
int main( )
{
    using namespace std;
    float a = 10.0F, b;
    b = a/2;
    cout << b << endl;
}
```

- a) 5.000000.
- b) 5.
- c) 5.0.
- d) 5.00000.

3) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
```

```
#include <iomanip>
int main( )
{
    using namespace std;
    double a = 20.0, b;
    b = a/3;
    cout << fixed << b << endl;
}
```

- a) Error. El formato tiene que ser %lf.
- b) 6.66666.
- c) 6.666666.
- d) 6.666667.

4) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    int a = 10;
    cout << hex << uppercase << a << endl;
}
```

- a) 10.
- b) 0xA.
- c) A.
- d) a.

5) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    double a = 20.0, b;
    b = a/3;
    cout << fixed << setprecision(2);
    cout << setw(20) << b << endl;
}
```

- a) 6.666666 ajustado al margen izquierdo de la pantalla.
- b) 6.666666 ajustado a la derecha en un ancho de 20 posiciones.
- c) 6.67 ajustado a la derecha en un ancho de 22 posiciones.
- d) Ninguno de los anteriores.

- 6) ¿Cuál es el resultado del siguiente programa suponiendo que se teclea el valor 3.1416?

```
#include <iostream>
int main( )
{
    using namespace std;
    long a = 0;
    cin >> a;
    cout << a << endl;
}
```

- a) 3.1416.
- b) 3.
- c) 3.142.
- d) 0.0.

- 7) ¿Cuál es el resultado del siguiente programa suponiendo que se teclea el valor 3.1416?

```
#include <iostream>
int main( )
{
    using namespace std;
    long a = 0;
    double b = 0.0;
    cin >> a >> b;
    cout << b << endl;
}
```

- a) 3.1416.
- b) Error durante la ejecución.
- c) 0.1416.
- d) 0.0.

- 8) ¿Cuál es el resultado del siguiente programa suponiendo que se teclean los valores: 3.1416 x6A -3.1?

```
#include <iostream>
#include <iomanip>
int main( )
{
    using namespace std;
    double a = 0.0, b = 0.0, c = 0.0;
    bool r = true;
    cin >> a >> b >> c;
    r = cin.rdstate() == ios::goodbit;
    cout << boolalpha << r << endl;
}
```

- a) true.
- b) false.
- c) 1.
- d) 0.

- 9) ¿Cuál es el resultado del siguiente programa suponiendo que el valor ASCII del 0 es 48 y que se teclea el valor 2?

```
#include <iostream>
int main( )
{
    using namespace std;
    char c = 0;
    cin.get(c);
    cout << c * 2 << endl;
}
```

- a) 100.
- b) 96.
- c) 4.
- d) Ninguno de los anteriores.

- 10) ¿Cuál es el resultado que se ve en la pantalla cuando se ejecuta el siguiente programa suponiendo que se teclean los valores: 2 s?

```
#include <iostream>
int main( )
{
    using namespace std;
    int a = 0; char c = 0;
    cin >> a; cin.get(c);
    cout << a << ' ' << c << endl;
}
```

- a) 2 s.
- b) 2.
- c) s.
- d) Ninguno de los anteriores.

2. Realizar un programa que calcule el volumen de una esfera, que viene dado por la fórmula:

$$v = \frac{4}{3} \pi r^3$$

3. Realizar un programa que pregunte el nombre, el año de nacimiento y el año actual, y dé como resultado:

Hola nombre, en el año 2030 cumplirás n años

4. Realizar un programa que evalúe el polinomio $p = 3x^5 - 5x^3 + 2x - 7$ y visualice el resultado con el siguiente formato:

Para $x = \text{valor}$, $3x^5 - 5x^3 + 2x - 7 = \text{resultado}$

5. Realizar el mismo programa anterior, pero empleando ahora coeficientes variables a , b y c . Piense ahora en objetos y escriba una clase *CEcuacion* con los atributos a , b y c y los métodos *establecerCoeficientes* y *valorEcuacion*, además de los constructores necesarios. La función **main** creará un objeto ecuación y visualizará su valor para unos coeficientes y un valor de x introducidos por el teclado.
6. Ejecute el siguiente programa, explique lo que ocurre y realice las modificaciones que sean necesarias para su correcto funcionamiento.

```
#include <iostream>
int main( )
{
    using namespace std;
    cout << "Introducir un carácter: "; cin.get(car);
    cout << car << endl;
    cout << "Introducir otro carácter: "; cin.get(car);
    cout << car << endl;
}
```

7. Indique qué resultado da el siguiente programa. A continuación ejecute el programa y compare los resultados.

```
#include <iostream>
int main( )
{
    using namespace std;
    char car1 = 'A', car2 = 65, car3 = 0;
    car3 = car1 + 'a' - 'A';
    cout << static_cast<int>(car3) << ' ' << car3 << endl;
    car3 = car2 + 32;
    cout << static_cast<int>(car3) << ' ' << car3 << endl;
}
```


SENTENCIAS DE CONTROL

Frecuentemente surge la necesidad de ejecutar unas sentencias u otras en función del valor que tomen una o más expresiones en un instante determinado durante la ejecución del programa. Así mismo, en más de una ocasión habrá que ejecutar un conjunto de sentencias un número determinado de veces, o bien hasta que se cumpla una determinada condición.

En este capítulo aprenderá a escribir código para que un programa tome decisiones y para que sea capaz de ejecutar bloques de sentencias repetidas veces.

SENTENCIA **if**

La sentencia **if** permite a un programa tomar una decisión para ejecutar una acción u otra, basándose en el resultado verdadero o falso de una expresión. La sintaxis para utilizar esta sentencia es la siguiente:

```
if (condición)
    sentencia 1;
[else
    sentencia 2];
```

donde *condición* es una expresión numérica, relacional o lógica, y *sentencia 1* y *sentencia 2* representan a una sentencia simple o compuesta. Cada sentencia simple debe finalizar con un punto y coma.

Una sentencia **if** se ejecuta de la forma siguiente:

1. Se evalúa la *condición* obteniéndose un resultado verdadero o falso.

2. Si el resultado es verdadero (resultado **true** o distinto de 0), se ejecutará lo indicado por la *sentencia 1*.
3. Si el resultado es falso (resultado **false** o 0), la *sentencia 1* se ignora y se ejecutará lo indicado por la *sentencia 2*, si la cláusula **else** se ha especificado.
4. En cualquier caso, la ejecución continúa en la siguiente sentencia ejecutable que haya a continuación de la sentencia **if**.

A continuación se exponen algunos ejemplos para que vea de una forma sencilla cómo se utiliza la sentencia **if**.

```
if (x)           // es lo mismo que: if (x != 0)
    b = a / x;
b = b + 1;
```

En este ejemplo, la condición viene impuesta por una expresión numérica x . Entonces $b = a/x$, que sustituye a la *sentencia 1* del formato general, se ejecutará si la expresión es verdadera (x distinta de 0) y no se ejecutará si la expresión es falsa (x igual a 0). En cualquier caso, se continúa la ejecución en la línea siguiente, $b = b + 1$. Veamos otro ejemplo:

```
if (a < b) c = c + 1;
// siguiente línea del programa
```

En este otro ejemplo, la condición viene impuesta por una expresión de relación. Si al evaluar la condición se cumple que a es menor que b , entonces se ejecuta la sentencia $c = c + 1$. En otro caso, esto es, si a es mayor o igual que b , se continúa en la línea siguiente, ignorándose la sentencia $c = c + 1$.

En el ejemplo siguiente, la condición viene impuesta por una expresión lógica. Si al evaluar la condición se cumple que a y b son distintas de 0, entonces se ejecuta la sentencia $x = i$. En otro caso, la sentencia $x = i$ se ignora, continuando la ejecución en la línea siguiente.

```
if (a && b)      // es lo mismo que: if (a != 0 && b != 0)
    x = i;
// siguiente línea del programa
```

En el ejemplo siguiente, si se cumple que a es igual a $b*5$, se ejecutan las sentencias $x = 4$ y $a = a + x$. En otro caso, se ejecuta la sentencia $b = 0$. En ambos casos, la ejecución continúa en la siguiente línea del programa.

```
if (a == b * 5)
{
    x = 4;
```

```

    a = a + x;
}
else
    b = 0;
// siguiente línea del programa

```

Un error típico es escribir, en lugar de la condición del ejemplo anterior, la siguiente:

```

if (a = b * 5)
// ...

```

que equivale a:

```

a = b * 5;
if (a) // es lo mismo que: if ((a = b * 5) != 0)
// ...

```

En este otro ejemplo que se muestra a continuación, la sentencia **return** se ejecutará solamente cuando *car* sea igual al carácter 's'.

```

if (car == 's')
    return;

```

ANIDAMIENTO DE SENTENCIAS if

Observando el formato general de la sentencia **if** cabe una pregunta: ¿se puede escribir otra sentencia **if** como *sentencia 1* o *sentencia 2*? La respuesta es sí. Esto es, las sentencias **if ... else** pueden estar anidadas. Por ejemplo:

```

if (condición 1)
{
    if (condición 2)
        sentencia 1;
}
else
    sentencia 2;

```

Al evaluarse las condiciones anteriores, pueden presentarse los casos que se indican en la tabla siguiente:

condición 1	condición 2	se ejecuta: sentencia 1	sentencia 2
F	F	no	sí
F	V	no	sí
V	F	no	no
V	V	sí	no

(V = verdadero, F = falso, no = no se ejecuta, sí = sí se ejecuta)

En el ejemplo anterior las llaves definen perfectamente que la cláusula **else** está emparejada con el primer **if**. ¿Qué sucede si quitamos las llaves?

```
if (condición 1)
    if (condición 2)
        sentencia 1;
    else
        sentencia 2;
```

Ahora podríamos dudar de a qué **if** pertenece la cláusula **else**. Cuando en el código de un programa aparecen sentencias **if ... else** anidadas, la regla para diferenciar cada una de estas sentencias es que “cada **else** se corresponde con el **if** más próximo que no haya sido emparejado”. Según esto la cláusula **else** está emparejada con el segundo **if**. Entonces, al evaluarse ahora las *condiciones 1 y 2*, pueden presentarse los casos que se indican en la tabla siguiente:

condición 1	condición 2	se ejecuta: sentencia 1	sentencia 2
F	F	no	no
F	V	no	no
V	F	no	sí
V	V	sí	no

(V = verdadero, F = falso, no = no se ejecuta, sí = sí se ejecuta)

Es importante observar que una vez que se ejecuta una acción como resultado de haber evaluado las condiciones impuestas, la ejecución del programa continúa en la siguiente línea a la estructura a que dan lugar las sentencias **if ... else** anidadas. Por ejemplo, si en el ejemplo siguiente ocurre que *a* no es igual a *0*, la ejecución continúa en la siguiente línea del programa.

```
if (a == 0)
    if (b != 0)
        s = s + b;
    else
        s = s + a;
// siguiente línea del programa
```

Si en lugar de la solución anterior, lo que deseamos es que se ejecute $s = s + a$ cuando a no es igual a 0, entonces tendremos que incluir entre llaves el segundo **if** sin la cláusula **else**; esto es:

```
if (a == 0)
{
    if (b != 0)
        s = s + b;
}
else
    s = s + a;
// siguiente línea del programa
```

Como ejercicio sobre la teoría expuesta, vamos a realizar una aplicación que dé como resultado el menor de tres números a , b y c . La forma de proceder es comparar cada número con los otros dos una sola vez. La simple lectura del código que se muestra a continuación es suficiente para entender el proceso seguido.

```
// if-else - Menor de tres números a, b y c
#include <iostream>

int main( )
{
    using namespace std;
    float a, b, c, menor;

    cout << "Números a b c : ";
    cin >> a >> b >> c;

    if (a < b)
        if (a < c)
            menor = a;
        else
            menor = c;
    else
        if (b < c)
            menor = b;
        else
            menor = c;

    cout << "Menor = " << menor << endl;
}
```

Ejecución del programa:

```
Números a b c : 25.84 -3.1 18
Menor = -3.1
```

ESTRUCTURA **else if**

La estructura presentada a continuación aparece con bastante frecuencia y es por lo que se le da un tratamiento por separado. Esta estructura es consecuencia de las sentencias **if** anidadas. Su formato general es:

```
if (condición 1)
    sentencia 1;
else if (condición 2)
    sentencia 2;
else if (condición 3)
    sentencia 3;
.
.
.
else
    sentencia n;
```

La evaluación de esta estructura sucede así: si se cumple la *condición 1*, se ejecuta la *sentencia 1* y si no se cumple, se examinan secuencialmente las condiciones siguientes hasta el último **else**, ejecutándose la sentencia correspondiente al primer **else if**, cuya *condición* sea cierta. Si todas las condiciones son falsas, se ejecuta la *sentencia n* correspondiente al último **else**. En cualquier caso, se continúa en la primera sentencia ejecutable que haya a continuación de la estructura. Las *sentencias 1, 2, ..., n* pueden ser sentencias simples o compuestas.

Por ejemplo, al efectuar una compra en un cierto almacén, si adquirimos más de 100 unidades de un mismo artículo, nos hacen un descuento de un 40%; entre 25 y 100 un 20%; entre 10 y 24 un 10%; y no hay descuento para una adquisición de menos de 10 unidades. Se pide calcular el importe a pagar. La solución se presentará de la siguiente forma:

```
Código artículo..... 111
Cantidad comprada..... 100
Precio unitario..... 100

Descuento..... 20%
Total..... 8000
```

En la solución presentada como ejemplo, se puede observar que como la cantidad comprada está entre 25 y 100, el descuento aplicado es de un 20%.

La solución de este problema puede ser de la forma siguiente:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int ar, cc; // código y cantidad
float pu; // precio unitario
float desc; // descuento
```

- A continuación leemos los datos *ar*, *cc* y *pu*.

```
cout << "Código artículo..... "; cin >> ar;
cout << "Cantidad comprada..... "; cin >> cc;
cout << "Precio unitario..... "; cin >> pu;
```

- Conocidos los datos, realizamos los cálculos y escribimos el resultado.

```
if (cc > 100)
    desc = 40.0F; // descuento 40%
else if (cc >= 25)
    desc = 20.0F; // descuento 20%
else if (cc >= 10)
    desc = 10.0F; // descuento 10%
else
    desc = 0.0F; // descuento 0%
cout << "Descuento..... " << desc << "%\n";
cout << "Total..... "
    << cc * pu * (1 - desc / 100) << '\n';
```

Se puede observar que las condiciones se han establecido según los descuentos de mayor a menor. Como ejercicio, piense o pruebe qué ocurriría si establece las condiciones según los descuentos de menor a mayor. El programa completo se muestra a continuación.

```
// else-if - Cantidad a pagar en función de la compra
#include <iostream>
```

```
int main( )
{
    using namespace std;
    int ar, cc; // código y cantidad
    float pu; // precio unitario
    float desc; // descuento

    cout << "Código artículo..... "; cin >> ar;
    cout << "Cantidad comprada..... "; cin >> cc;
    cout << "Precio unitario..... "; cin >> pu;
    cout << endl;

    if (cc > 100)
        desc = 40.0F; // descuento 40%
    else if (cc >= 25)
        desc = 20.0F; // descuento 20%
    else if (cc >= 10)
```

```

    desc = 10.0F;      // descuento 10%
else
    desc = 0.0F;      // descuento 0%
cout << "Descuento..... " << desc << "%\n";
cout << "Total..... "
    << cc * pu * (1 - desc / 100) << '\n';
}

```

SENTENCIA **switch**

La sentencia **switch** permite ejecutar una de varias acciones, en función del valor de una expresión. Es una sentencia especial para decisiones múltiples. La sintaxis para utilizar esta sentencia es:

```

switch (expresión)
{
    [case expresión-constante 1:]
        [sentencia 1;]
    [case expresión-constante 2:]
        [sentencia 2;]
    [case expresión-constante 3:]
        [sentencia 3;]
    .
    .
    .
    [default:]
        [sentencia n;]
}

```

donde *expresión* es una expresión de tipo entero o enumerado y *expresión-constante* es una constante del mismo tipo que *expresión* o de un tipo que se pueda convertir implícitamente al tipo de *expresión*; y *sentencia* es una sentencia simple o compuesta. En el caso de tratarse de una sentencia compuesta, no hace falta incluir las sentencias simples que la forman entre `{}`.

La sentencia **switch** evalúa la expresión entre paréntesis y compara su valor con las constantes de cada **case**. La ejecución de las sentencias del bloque de la sentencia **switch** comienza en el **case** cuya constante coincida con el valor de la expresión y continúa hasta una sentencia que transfiera el control dentro o fuera del bloque de **switch**; esta sentencia debe estar presente por cada **case** así como para **default**. Generalmente se utiliza **break** para transferir el control fuera del bloque de la sentencia **switch**. La sentencia **switch** puede incluir cualquier número de cláusulas **case** y una cláusula **default** como mucho.

Si no existe una constante igual al valor de la expresión, entonces se ejecutan las sentencias que están a continuación de **default**, si esta cláusula ha sido especi-

ficada. La cláusula **default** puede colocarse en cualquier parte del bloque y no necesariamente al final.

Para ilustrar la sentencia **switch**, vamos a realizar un programa que lea una fecha representada por dos enteros, *mes* y *año*, y dé como resultado los días correspondientes al *mes*. Esto es:

```
Mes (##): 5
Año (####): 2014
```

```
El mes 5 del año 2014 tiene 31 días
```

Hay que tener en cuenta que febrero puede tener 28 días, o bien 29 si el año es bisiesto. Un año es bisiesto cuando es múltiplo de 4 y no de 100 o cuando es múltiplo de 400. Por ejemplo, el año 2000 por las dos primeras condiciones no sería bisiesto, pero sí lo es porque es múltiplo de 400; el año 2100 no es bisiesto porque aunque sea múltiplo de 4, también lo es de 100 y no es múltiplo de 400.

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int dd = 0, mm = 0, aa = 0;
```

- A continuación leemos los datos *mes* (*mm*) y *año* (*aa*).

```
cout << "Mes (##): "; cin >> mm;
cout << "Año (####): "; cin >> aa;
```

- Después comparamos el *mes* con las constantes 1, 2, ..., 12. Si *mes* es 1, 3, 5, 7, 8, 10 ó 12, asignamos a *días* el valor 31. Si *mes* es 4, 6, 9 u 11, asignamos a *días* el valor 30. Si *mes* es 2, verificaremos si el *año* es bisiesto, en cuyo caso asignamos a *días* el valor 29 y si no es bisiesto, asignamos a *días* el valor 28. Si *mes* no es ningún valor de los anteriores, enviaremos un mensaje al usuario indicándole que el mes no es válido. Todo este proceso lo realizaremos con una sentencia **switch**.

```
switch (mm)
{
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        dd = 31;
        break;
    case 4: case 6: case 9: case 11:
        dd = 30;
        break;
    case 2:
```

```

// ¿Es el año bisiesto?
if ((aa % 4 == 0) && (aa % 100 != 0) || (aa % 400 == 0))
    dd = 29;
else
    dd = 28;
    break;
default:
    cout << "\nEl mes no es válido\n";
    break;
}

```

Cuando una constante coincide con el valor de *mm*, se ejecutan las sentencias especificadas a continuación de la misma, siguiendo la ejecución del programa por los bloques de las siguientes cláusulas **case**, a no ser que se tome una acción explícita para abandonar el bloque de la sentencia **switch**. Ésta es precisamente la función de la sentencia **break** al final de cada bloque **case**.

- Por último si el *mes* es válido, escribimos el resultado solicitado.

```

if (mes >= 1 && mes <= 12)
    cout << "\nEl mes " << mm << " del año " << aa
        << " tiene " << dd << " días" << '\n';

```

El programa completo se muestra a continuación:

```

// switch - Días correspondientes a un mes de un año dado
#include <iostream>

int main( )
{
    using namespace std;
    int dd = 0, mm = 0, aa = 0;

    cout << "Mes (##): "; cin >> mm;
    cout << "Año (####): "; cin >> aa;

    switch (mm)
    {
        case 1:        // enero
        case 3:        // marzo
        case 5:        // mayo
        case 7:        // julio
        case 8:        // agosto
        case 10:       // octubre
        case 12:       // diciembre
            dd = 31;
            break;
        case 4:        // abril
        case 6:        // junio

```

```

case 9:      // septiembre
case 11:     // noviembre
    dd = 30;
    break;
case 2:      // febrero
    // ¿Es el año bisiesto?
    if ((aa % 4 == 0) && (aa % 100 != 0) || (aa % 400 == 0))
        dd = 29;
    else
        dd = 28;
    break;
default:
    cout << "\nEl mes no es válido\n";
    break;
}
if (mm >= 1 && mm <= 12)
    cout << "\nEl mes " << mm << " del año " << aa
        << " tiene " << dd << " días" << '\n';
}

```

El que las cláusulas **case** estén una a continuación de otra o una debajo de otra no es más que una cuestión de estilo, ya que C++ interpreta cada carácter nueva línea como un espacio en blanco; esto es, el código al que llega el compilador es el mismo en cualquier caso.

La sentencia **break** que se ha puesto a continuación de la cláusula **default** no es necesaria; simplemente obedece a un buen estilo de programación. Así, cuando tengamos que añadir otro caso ya tenemos puesto **break**, con lo que hemos eliminado una posible fuente de errores.

SENTENCIA **while**

La sentencia **while** ejecuta una sentencia, simple o compuesta, cero o más veces, dependiendo de una expresión. Su sintaxis es:

```

while (expresión)
    sentencia;

```

donde *expresión* es cualquier expresión numérica, relacional o lógica, y *sentencia* es una sentencia simple o compuesta.

La ejecución de la sentencia **while** sucede así:

1. Se evalúa la *expresión* obteniéndose un resultado verdadero o falso.
2. Si el resultado es falso (resultado **false** o 0), la sentencia no se ejecuta y se pasa el control a la siguiente sentencia en el programa.

3. Si el resultado es verdadero (resultado **true** o distinto de 0), se ejecuta la sentencia y el proceso descrito se repite desde el punto 1.

Por ejemplo, el siguiente código, que podrá ser incluido en cualquier programa, solicita obligatoriamente una de las dos respuestas posibles: *s/n* (sí o no).

```
#include <iostream>

int main( )
{
    using namespace std;
    char car = '\0';

    cout << "Desea continuar s/n (sí o no) ";
    cin.get(car);
    while (car != 's' && car != 'n')
    {
        cin.ignore();
        cout << "Desea continuar s/n (sí o no) ";
        cin.get(car);
    }
}
```

Ejecución del programa:

```
Desea continuar s/n (sí o no) x
Desea continuar s/n (sí o no) c
Desea continuar s/n (sí o no) n
```

Observe que antes de ejecutarse la sentencia **while** se visualiza el mensaje “Desea continuar s/n (sí o no)” y se inicia la expresión; esto es, se asigna un carácter a la variable *car* que interviene en la expresión de la sentencia **while**.

La sentencia **while** se interpreta de la forma siguiente: mientras el valor de *car* no sea igual ni al carácter ‘s’ ni al carácter ‘n’, visualizar el mensaje “Desea continuar s/n (sí o no)” y leer otro carácter. Esto obliga al usuario a escribir el carácter ‘s’ o ‘n’ en minúsculas.

El siguiente ejemplo visualiza el código ASCII de cada uno de los caracteres de una cadena de texto introducida por el teclado. En este caso, la sentencia **while** evalúa dos expresiones: leer un carácter y verificar si hay más caracteres para leer, pero sólo la segunda expresión interviene en la evaluación de la condición (véase el operador coma en el capítulo 2). En este caso, la condición de terminación es leer datos hasta alcanzar la marca de fin de fichero. Recuerde que para el flujo estándar de entrada, esta marca se produce cuando se pulsan las teclas *Ctrl+D* en UNIX, o bien *Ctrl+Z* en aplicaciones Windows de consola, y que cuando **get** lee

una marca de fin de fichero, pone el estado del flujo **cin** al valor **eofbit** (también se activa **failbit** porque la entrada no es válida).

```
// Código ASCII (0 a 255) de los caracteres de un texto
#include <iostream>

int main( )
{
    using namespace std;
    char car = 0; // car = carácter nulo (\0)
    unsigned char ucar = 0;

    cout << "Introduzca una cadena de texto.\n";
    cout << "Para terminar pulse Ctrl+z\n";
    while (cin.get(car), !cin.eof())
    {
        if (car != '\n')
        {
            ucar = car; // convertirlo a un entero sin signo
            cout << "El código ASCII de " << car << " es "
                << static_cast<int>(ucar) << endl;
        }
    }
    cin.clear(); // estado igual goodbit
}
```

Una ejecución de este programa puede ser la siguiente:

```
Introduzca una cadena de texto.
Para terminar pulse Ctrl+z
```

```
hola[Entrar]
El código ASCII de h es 104
El código ASCII de o es 111
El código ASCII de l es 108
El código ASCII de a es 97
adiós[Entrar]
El código ASCII de a es 97
El código ASCII de d es 100
El código ASCII de i es 105
El código ASCII de ó es 162
El código ASCII de s es 115
[Ctrl] [z]
```

Este resultado demuestra que sólo se visualiza el código ASCII de los caracteres que hay hasta la pulsación *Entrar*; el carácter *\n* introducido al pulsar *Entrar* es ignorado porque así se ha programado. Cuando se han leído todos los caracteres del flujo de entrada, se solicitan nuevos datos. Lógicamente, habrá compren-

dido que aunque se lea carácter a carácter se puede escribir (almacenar en el búfer de entrada), hasta pulsar *Entrar*, un texto cualquiera.

Bucles anidados

Cuando se incluye una sentencia **while** dentro de otra sentencia **while**, en general una sentencia **while**, **do** o **for** dentro de otra de ellas, estamos en el caso de bucles anidados. Por ejemplo:

```
// Bucles anidados
#include <iostream>

int main( )
{
    using namespace std;
    int i = 1, j = 1;

    while ( i <= 3 ) // mientras i sea menor o igual que 3
    {
        cout << "Para i = " << i << ": ";
        while ( j <= 4 ) // mientras j sea menor o igual que 4
        {
            cout << "j = " << j << ", ";
            j++; // aumentar j en una unidad
        }
        cout << endl; // avanzar a una nueva línea
        i++; // aumentar i en una unidad
        j = 1; // iniciar j de nuevo a 1
    }
}
```

Al ejecutar este programa se obtiene el siguiente resultado:

```
Para i = 1: j = 1, j = 2, j = 3, j = 4,
Para i = 2: j = 1, j = 2, j = 3, j = 4,
Para i = 3: j = 1, j = 2, j = 3, j = 4,
```

Este resultado demuestra que el bucle exterior se ejecuta tres veces, y por cada una de éstas, el bucle interior se ejecuta a su vez cuatro veces. Es así como se ejecutan los bucles anidados: por cada iteración del bucle externo, el interno se ejecuta hasta finalizar todas sus iteraciones.

Observe también que cada vez que finaliza la ejecución de la sentencia **while** interior, avanzamos a una nueva línea, incrementamos el valor de *i* en una unidad e iniciamos de nuevo *j* al valor 1.

Como aplicación de lo expuesto, vamos a realizar un programa que imprima los números z , comprendidos entre 1 y 50, que cumplan la expresión:

$$z^2 = x^2 + y^2$$

donde z , x e y son números enteros positivos. El resultado se presentará de la forma siguiente:

Z	X	Y
5	3	4
13	5	12
10	6	8
...
50	30	40

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
unsigned int x = 1, y = 1, z = 0;
```

- A continuación escribimos la cabecera de la solución.

```
cout << "Z\t" << "X\t" << "Y\n";
cout << "_____\n";
```

- Después, para $x = 1$, e $y = 1, 2, 3, \dots$, para $x = 2$, e $y = 2, 3, 4, \dots$, para $x = 3$, e $y = 3, 4, \dots$, hasta $x = 50$, calculamos $\sqrt{x^2 + y^2}$; llamamos a este valor z (observe que y es igual o mayor que x para evitar que se repitan pares de valores como $x=3, y=4$ y $x=4, y=3$). Si z es exacto, escribimos z, x e y . Esto es, para los valores descritos de x e y , hacemos los cálculos:

```
z = static_cast<int>(sqrt(x * x + y * y)); // z es entera
if (z * z == x * x + y * y) // ¿la raíz cuadrada fue exacta?
    cout << z << "\t" << x << "\t" << y << "\n";
```

Además, siempre que obtengamos un valor z mayor que 50 lo desecharemos y continuaremos con un nuevo valor de x y los correspondientes valores de y .

El programa completo se muestra a continuación:

```
// Teorema de Pitágoras.
#include <iostream>
#include <cmath>
```

```
int main( )
{
    using namespace std;
    unsigned int x = 1, y = 1, z = 0;

    cout << "Z\t" << "X\t" << "Y\n";
    cout << "_____ \n";
    while (x <= 50)
    {
        // Calcular z. Como z es un entero, almacena
        // la parte entera de la raíz cuadrada
        z = static_cast<int>(sqrt(x * x + y * y));
        while (y <= 50 && z <= 50)
        {
            // Si la raíz cuadrada anterior fue exacta,
            // escribir z, x e y
            if (z * z == x * x + y * y)
                cout << z << "\t" << x << "\t" << y << "\n";
            y = y + 1;
            z = static_cast<int>(sqrt(x * x + y * y));
        }
        x = x + 1; y = x;
    }
}
```

SENTENCIA **do ... while**

La sentencia **do ... while** ejecuta una sentencia, simple o compuesta, una o más veces dependiendo del valor de una expresión. Su sintaxis es la siguiente:

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

donde *expresión* es cualquier expresión numérica, relacional o lógica, y *sentencia* es una sentencia simple o compuesta. Observe que la estructura **do ... while** finaliza con un punto y coma.

La ejecución de una sentencia **do ... while** sucede de la siguiente forma:

1. Se ejecuta el bloque (sentencia simple o compuesta) de **do**.
2. Se evalúa la *expresión* correspondiente a la condición de finalización del bucle obteniéndose un resultado verdadero o falso.
3. Si el resultado es falso (resultado **false** o 0), se pasa el control a la siguiente sentencia en el programa.

4. Si el resultado es verdadero (resultado **true** o distinto de 0), el proceso descrito se repite desde el punto 1.

Por ejemplo, el siguiente código obliga al usuario a introducir un valor positivo:

```
double n;
do // ejecutar las sentencias siguientes
{
    cout << "Número: ";
    cin >> n;
}
while ( n < 0 ); // mientras n sea menor que 0
```

Cuando se utiliza una sentencia **do ... while** el bloque de sentencias se ejecuta al menos una vez, porque la condición de terminación se evalúa al final. En cambio, cuando se ejecuta una sentencia **while** puede suceder que el bloque de sentencias no se ejecute, lo que ocurrirá siempre que la condición de terminación sea inicialmente falsa.

Como ejercicio, vamos a realizar un programa que calcule la raíz cuadrada de un número n por el método de Newton. Este método se enuncia así: sea r_i la raíz cuadrada aproximada de n . La siguiente raíz aproximada r_{i+1} se calcula en función de la anterior así:

$$r_{i+1} = \frac{\frac{n}{r_i} + r_i}{2}$$

El proceso descrito se repite hasta que la diferencia en valor absoluto de las dos últimas aproximaciones calculadas sea tan pequeña como nosotros queramos (teniendo en cuenta los límites establecidos por el tipo de datos utilizado). Según esto, la última aproximación será una raíz válida cuando se cumpla que:

$$\text{abs}(r_i - r_{i+1}) \leq \varepsilon$$

La solución de este problema puede ser de la siguiente forma:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
double n;           // número
double aprox;      // aproximación a la raíz cuadrada
double antaprox;   // anterior aproximación a la raíz cuadrada
double epsilon;    // coeficiente de error
```

- A continuación leemos los datos n , $aprox$ y $epsilon$.

```
cout << "Número: ";
cin >> n;
cout << "Raíz cuadrada aproximada: ";
cin >> aprox;
cout << "Coeficiente de error: ";
cin >> epsilon;
```

- Después, aplicamos la fórmula de Newton.

```
do
{
    antaprox = aprox;
    aprox = (n/antaprox + antaprox) / 2;
}
while (fabs(aprox - antaprox) >= epsilon);
```

Al aplicar la fórmula por primera vez, la variable *antaprox* contiene el valor aproximado a la raíz cuadrada que hemos introducido a través del teclado. Para sucesivas veces, *antaprox* contendrá la última aproximación calculada.

- Cuando la condición especificada en la estructura **do ... while** mostrada anteriormente sea falsa, el proceso habrá terminado. Sólo queda imprimir el resultado.

```
cout << "La raíz cuadrada de " << n << " es " << aprox << endl;
```

El programa completo se muestra a continuación. Para no permitir la entrada de número negativos, se ha utilizado una sentencia **do ... while** que preguntará por el valor solicitado mientras el introducido sea negativo.

```
// Raíz cuadrada de un número. Método de Newton.
#include <iostream>
#include <cmath>
#include <iomanip>

int main( )
{
    using namespace std;
    double n;          // número
    double aprox;     // aproximación a la raíz cuadrada
    double antaprox;  // anterior aproximación a la raíz cuadrada
    double epsilon;   // coeficiente de error
    do
    {
        cout << "Número: ";
        cin >> n;
    }
    while ( n < 0 );
```

```

do
{
    cout << "Raíz cuadrada aproximada: ";
    cin >> aprox;
}
while ( aprox <= 0 );

do
{
    cout << "Coeficiente de error: ";
    cin >> epsilon;
}
while ( epsilon <= 0 );

do
{
    antaprox = aprox;
    aprox = (n/antaprox + antaprox) / 2;
}
while (fabs(aprox - antaprox) >= epsilon);

cout << fixed << setprecision(2);
cout << "La raíz cuadrada de " << n << " es " << aprox << endl;
}

```

Si ejecuta este programa para un valor de n igual a 10, obtendrá la siguiente solución:

```

Número: 10
Raíz cuadrada aproximada: 1
Coeficiente de error: 1e-4
La raíz cuadrada de 10.00 es 3.16

```

SENTENCIA for

La sentencia **for** permite ejecutar una sentencia simple o compuesta, repetidamente un número de veces conocido. Su sintaxis es la siguiente:

```

for ([v1=e1 [, v2=e2]...];[condición];[progresión-condición])
    sentencia;

```

- $v1$, $v2$, ... representan variables de control que serán iniciadas con los valores de las expresiones $e1$, $e2$, ...;
- *condición* es cualquier expresión numérica, relacional o lógica que se evalúa a un valor verdadero o falso; si se omite, se supone verdadera;

- *progresión-condición* es una o más expresiones separadas por comas cuyos valores evolucionan en el sentido de que se cumpla la condición para finalizar la ejecución de la sentencia **for**;
- *sentencia* es una sentencia simple o compuesta.

La ejecución de la sentencia **for** sucede de la siguiente forma:

1. Se inician las variables $v1, v2, \dots$
2. Se evalúa la condición de finalización del bucle obteniéndose un resultado verdadero o falso:
 - a) Si el resultado es verdadero (resultado **true** o distinto de 0), se ejecuta el bloque de sentencias, se evalúa la expresión que da lugar a la progresión de la condición y se vuelve al punto 2.
 - b) Si el resultado es falso (resultado **false** o 0), la ejecución de la sentencia **for** se da por finalizada y se pasa el control a la siguiente sentencia del programa.

Por ejemplo, la siguiente sentencia **for** imprime los números del 1 al 100. Literalmente dice: desde i igual a 1, mientras i sea menor o igual que 100, incrementando la i de uno en uno, escribir el valor de i .

```
int i;
for (i = 1; i <= 100; i++)
  cout << i << ' ';
```

El siguiente ejemplo imprime los múltiplos de 7 que hay entre 7 y 112. Se puede observar que, en este caso, la variable se ha declarado e iniciado en la propia sentencia **for** por lo que tan sólo puede utilizarse en el mismo (esto no se puede hacer en una sentencia **while**; las variables que intervienen en la condición de una sentencia **while** deben haber sido declaradas e iniciadas antes de que se procese la condición por primera vez).

```
for (int k = 7; k <= 112; k += 7)
  cout << k << " ";
```

En el ejemplo siguiente se puede observar la utilización de la coma como separador de las variables de control y de las expresiones que hacen que evolucionen los valores que intervienen en la condición de finalización.

```
int f, c;
for (f = 3, c = 6; f + c < 40; f++, c += 2)
  cout << "f = " << f << "\tc = " << c << endl;
```

Este otro ejemplo que ve a continuación imprime los valores desde 1 hasta 10 con incrementos de 0.5.

```
for (float i = 1; i <= 10; i += 0.5F)
    cout << i << ' ';
```

El siguiente ejemplo imprime las letras del abecedario en orden inverso.

```
char car;
for (car = 'z'; car >= 'a'; car--)
    cout << car << ' ';
```

El ejemplo siguiente indica cómo realizar un bucle infinito. Para salir de un bucle infinito tiene que pulsar las teclas *Ctrl+C*, o bien ejecutar **break**.

```
for (;;)
{
    sentencias;
}
```

Como aplicación de la sentencia **for** vamos a imprimir un tablero de ajedrez en el que las casillas blancas se simbolizarán con una B y las negras con una N. Así mismo, el programa deberá marcar con * las casillas a las que se puede mover un alfil desde una posición dada. La solución será similar a la siguiente:

```
Posición del alfil:
    fila      3
    columna   4
```

```
B * B N B * B N
N B * B * B N B
B N B * B N B N
N B * B * B N B
B * B N B * B N
* B N B N B * B
B N B N B N B *
N B N B N B N B
```

Desarrollo del programa:

- Primero definimos las variables que vamos a utilizar en los cálculos.

```
int falfil, calfil; // posición inicial del alfil
int fila, columna; // posición actual del alfil
```

- Leer la fila y la columna en la que se coloca el alfil.

```
cout << "Posición del alfil:\n";
cout << "  fila    "; cin >> falfil;
cout << "  columna "; cin >> calfil;
```

- Partiendo de la fila 1, columna 1 y recorriendo el tablero por filas,

```
for (fila = 1; fila <= 8; fila++)
{
    for (columna = 1; columna <= 8; columna++)
    {
        // Pintar el tablero de ajedrez
    }
    cout << endl; // cambiar de fila
}
```

imprimir un *, una B o una N dependiendo de las condiciones especificadas a continuación:

- ◇ Imprimir un * si se cumple que la suma o diferencia de la fila y columna actuales coincide con la suma o diferencia de la fila y columna donde se coloca el alfil.
- ◇ Imprimir una B si se cumple que la fila más columna actuales es par.
- ◇ Imprimir una N si se cumple que la fila más columna actuales es impar.

```
// Pintar el tablero de ajedrez
if ((fila + columna == falfil + calfil) ||
    (fila - columna == falfil - calfil))
    cout << "* ";
else if ((fila + columna) % 2 == 0)
    cout << "B ";
else
    cout << "N ";
```

El programa completo se muestra a continuación.

```
// Tablero de ajedrez
#include <iostream>

int main( )
{
    using namespace std;
    int falfil, calfil; // posición del alfil
    int fila, columna; // posición actual

    cout << "Posición del alfil:\n";
    cout << "  fila    "; cin >> falfil;
    cout << "  columna "; cin >> calfil;
    cout << endl; // dejar una línea en blanco
```

```

// Pintar el tablero de ajedrez
for (fila = 1; fila <= 8; fila++)
{
    for (columna = 1; columna <= 8; columna++)
    {
        if ((fila + columna == falfil + calfil) ||
            (fila - columna == falfil - calfil))
            cout << "* ";
        else if ((fila + columna) % 2 == 0)
            cout << "B ";
        else
            cout << "N ";
    }
    cout << endl; // cambiar de fila
}
}

```

SENTENCIA break

Anteriormente vimos que la sentencia **break** finaliza la ejecución de una sentencia **switch**. Pues bien, cuando se utiliza **break** en el bloque correspondiente a una sentencia **while**, **do** o **for**, hace lo mismo: finaliza la ejecución del bucle.

Cuando las sentencias **switch**, **while**, **do** o **for** estén anidadas, la sentencia **break** solamente finaliza la ejecución del bucle donde esté incluida.

Por ejemplo, el bucle del programa “*Código ASCII...*” desarrollado anteriormente podría escribirse también así:

```

// Código ASCII de los caracteres de un texto
#include <iostream>

int main( )
{
    using namespace std;
    char car = 0; // car = carácter nulo (\0)
    unsigned char ucar = 0;

    cout << "Introduzca una cadena de texto.\n";
    cout << "Para terminar pulse Ctrl+z\n";
    while (true)
    {
        cin.get(car);
        if (cin.eof()) break;
        if (car != '\n')
        {
            ucar = car; // convertirlo a un entero sin signo

```

```
        cout << "El código ASCII de " << car << " es "  
            << static_cast<int>(ucar) << endl;  
    }  
}  
cin.clear(); // estado igual goodbit  
}
```

SENTENCIA **continue**

La sentencia **continue** obliga a ejecutar la siguiente iteración del bucle **while**, **do** o **for**, en el que está contenida. Su sintaxis es:

```
continue;
```

Como ejemplo, vea el siguiente programa que imprime todos los números entre 1 y 100 que son múltiplos de 5.

```
#include <iostream>  
  
int main( )  
{  
    using namespace std;  
  
    int n;  
    for (n = 0; n <= 100; n++)  
    {  
        // Si n no es múltiplo de 5, siguiente iteración  
        if (n % 5 != 0) continue;  
        // Imprime el siguiente múltiplo de 5  
        cout << n << ' '  
    }  
}
```

Ejecute este programa y observe que cada vez que se ejecuta la sentencia **continue**, se inicia la ejecución del bloque de sentencias de **for** para un nuevo valor de *n*.

SENTENCIA **goto**

La sentencia **goto** transfiere el control a una línea específica del programa, identificada por una *etiqueta*. Su sintaxis es la siguiente:

```
goto etiqueta;  
.  
.  
.  
etiqueta: sentencia;
```


Si la línea a la que se transfiere el control es una sentencia ejecutable, se ejecuta esa sentencia y las que le siguen. Si no es ejecutable, la ejecución se inicia en la primera sentencia ejecutable que se encuentre a continuación de dicha línea.

No se puede transferir el control fuera del cuerpo de la función en la que nos encontremos.

Un uso abusivo de esta sentencia da lugar a programas difíciles de interpretar y de mantener. Por ello, se utiliza solamente en ocasiones excepcionales. La función que desempeña una sentencia **goto** puede suplirse utilizando las sentencias **if...else**, **do**, **for**, **switch** o **while**.

El uso más normal consiste en abandonar la ejecución de alguna estructura profundamente anidada, cosa que no puede hacerse mediante la sentencia **break**, ya que ésta se limita únicamente a un solo nivel de anidamiento.

El siguiente ejemplo muestra cómo se utiliza la sentencia **goto**. Consta de dos bucles **for** anidados. En el bucle interior hay una sentencia **goto** que se ejecutará si se cumple la condición especificada. Si se ejecuta la sentencia **goto**, el control es transferido a la primera sentencia ejecutable que haya a continuación de la etiqueta *salir*.

```
// Utilización de la sentencia goto
#include <iostream>

int main( )
{
    using namespace std;
    const int K = 8;
    int f, c, n;

    cout << "Valor de n: ";
    cin >> n;
    for (f = 0; f < K; f++)
    {
        for (c = 0; c < K; c++)
        {
            if (f*c > n) goto salir;
        }
    }
    salir:
    if (f < K && c < K)
        cout << "(" << f << ", " << c << ")\n";
}
```

Ejecución del programa

Valor de n : 20
(3, 7)

SENTENCIAS `try ... catch`

Las “excepciones” son el medio que tiene C++ para separar la notificación de un error del tratamiento del mismo.

Por ejemplo, el método `at` de la plantilla `vector` de la biblioteca de C++ que estudiaremos más adelante proporciona verificación de índices fuera de rango; esto es, la expresión `v.at(i)` permite acceder al elemento i del vector v y lanzará una excepción `out_of_range` siempre que i esté fuera del rango del vector (las excepciones están declaradas en el fichero de cabecera `<stdexcept>`).

Según lo expuesto, el método `at` se limita a notificar que ha ocurrido un error lanzando una excepción y es el usuario del vector el que debe escribir el código para atraparla y manejarla. Para ello, C++ añade las palabras reservadas `try`, `catch` y `throw`.

El código que puede lanzar una excepción se encierra en un bloque `try`, la excepción se lanza con `throw` y se atrapa y se maneja en un bloque `catch`. Por ejemplo:

```
int main( )
{
    // ...
    try
    {
        // Código que puede lanzar una excepción.
    }
    catch(excepción 1)
    {
        // Se produjo la excepción 1. Se atrapa y se maneja.
    }
    catch(excepción 2)
    {
        // Se produjo la excepción 2. Se atrapa y se maneja.
    }
    // ...
}
```

La idea básica es que una función que encuentra un problema que no puede resolver (por ejemplo, acceder a un elemento fuera de los límites de un vector), eleva (`throw`) una excepción, con la esperanza de que quien la llamó directa o indirectamente pueda tratar el problema. Esta otra función deberá encerrar el código que puede lanzar excepciones en un bloque `try` y atraparlas en un bloque `catch`

para manejarlas. Cuando el bloque **catch** finaliza, la excepción se considera manejada y la ejecución continúa. Más adelante dedicaremos un capítulo al estudio de excepciones.

EJERCICIOS RESUELTOS

1. Realizar un programa que a través de un menú permita realizar las operaciones de *sumar*, *restar*, *multiplicar*, *dividir* y *salir*. Las operaciones constarán solamente de dos operandos. El menú será visualizado por una función sin argumentos, que devolverá como resultado la opción elegida. La ejecución será de la forma siguiente:

```
1. sumar
2. restar
3. multiplicar
4. dividir
5. salir
```

```
Seleccione la operación deseada: 3
Dato 1: 2.5
Dato 2: 10
Resultado = 25
Pulse [Entrar] para continuar
```

La solución de este problema puede ser de la siguiente forma:

- Declaramos las funciones que van a intervenir en el programa.

```
int menu(void);
double leerDato();
```

- Definimos las variables que van a intervenir en el programa.

```
double dato1 = 0, dato2 = 0, resultado = 0;
int operación = 0;
```

- A continuación presentamos el menú en la pantalla para poder elegir la operación a realizar.

```
operacion = menu();
```

La definición de la función *menu* puede ser así:

```
int menu()
{
    int op;
```

```
do
{
    cout << "\t1. sumar\n";
    cout << "\t2. restar\n";
    cout << "\t3. multiplicar\n";
    cout << "\t4. dividir\n";
    cout << "\t5. salir\n";
    cout << "\nSeleccione la operación deseada: ";
    op = static_cast<int>(leerDato());
}
while (op < 1 || op > 5);
return op;
}
```

- Si la operación elegida no ha sido *salir*, leemos los operandos *dato1* y *dato2*.

```
if (operacion != 5)
{
    // Leer datos
    cout << "Dato 1: "; dato1 = leerDato();
    cout << "Dato 2: "; dato2 = leerDato();

    // Realizar la operación
}
else
    break; // salir
```

La definición de la función *leerDato* puede ser así:

```
double leerDato()
{
    double dato = 0.0;

    cin >> dato;
    while (cin.fail()) // si el dato es incorrecto, limpiar el
    { // búfer y volverlo a leer
        cout << '\a';
        cin.clear();
        cin.ignore(numeric_limits<int>::max(), '\n');
        cin >> dato;
    }
    // Eliminar posibles caracteres sobrantes
    cin.ignore(numeric_limits<int>::max(), '\n')
    return dato;
}
```

- A continuación, realizamos la operación elegida con los datos leídos e imprimimos el resultado.

```

switch (operacion)
{
    case 1:
        resultado = dato1 + dato2;
        break;
    case 2:
        resultado = dato1 - dato2;
        break;
    case 3:
        resultado = dato1 * dato2;
        break;
    case 4:
        if (dato2 == 0)
            cout << "el divisor no puede ser 0\n";
        else
            resultado = dato1 / dato2;
        break;
}
// Escribir el resultado
cout << "Resultado = " << resultado << endl;
// Hacer una pausa
cout << "Pulse <Entrar> para continuar ";
cin.ignore(); // como el búfer está limpio, espera por un carácter

```

- Las operaciones descritas formarán parte de un bucle infinito formado por una **sentencia while** con el fin de poder encadenar distintas operaciones.

```

while (true)
{
    // sentencias
}

```

El programa completo se muestra a continuación.

```

// Simulación de una calculadora
#include <iostream>
#include <limits>

using namespace std;

// Declaración de funciones
int menu(void);
double leerDato();

int main()
{
    double dato1 = 0, dato2 = 0, resultado = 0;
    int operacion = 0;

```

```
while (true)
{
    operacion = menu();
    if (operacion != 5)
    {
        // Leer datos
        cout << "Dato 1: "; dato1 = leerDato();
        cout << "Dato 2: "; dato2 = leerDato();

        // Realizar la operación
        switch (operacion)
        {
            case 1:
                resultado = dato1 + dato2;
                break;
            case 2:
                resultado = dato1 - dato2;
                break;
            case 3:
                resultado = dato1 * dato2;
                break;
            case 4:
                if (dato2 == 0)
                    cout << "el divisor no puede ser 0\n";
                else
                    resultado = dato1 / dato2;
                break;
        }

        // Escribir el resultado
        cout << "Resultado = " << resultado << endl;
        // Hacer una pausa
        cout << "Pulse <Entrar> para continuar ";
        cin.ignore(); // como el búfer está limpio, espera por un carácter
    }
    else
        break;
}

int menu()
{
    int op;

    do
    {
        cout << "\t1. sumar\n";
        cout << "\t2. restar\n";
        cout << "\t3. multiplicar\n";
        cout << "\t4. dividir\n";
```

```

    cout << "\t5. salir\n";
    cout << "\nSeleccione la operación deseada: ";
    op = static_cast<int>(leerDato());
}
while (op < 1 || op > 5);
return op;
}

double leerDato()
{
    double dato = 0.0;
    cin >> dato;
    while (cin.fail()) // si el dato es incorrecto, limpiar el
    { // búfer y volverlo a leer
        cout << '\a';
        cin.clear();
        cin.ignore(numeric_limits<int>::max(), '\n');
        cin >> dato;
    }
    // Eliminar posibles caracteres sobrantes
    cin.ignore(numeric_limits<int>::max(), '\n');
    return dato;
}

```

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

1) ¿Cuál es el resultado del siguiente programa?

```

#include <iostream>

int main( )
{
    using namespace std;
    int x = 1, y = 1;
    if (x = y * 5)
        x = 0;
    else
        x = -1;
    cout << x << endl;
}

```

- a) 1.
- b) 5.
- c) 0.
- d) -1.

2) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

int main( )
{
    using namespace std;
    int x = 1, y = 1;
    if (x == 1)
        if (y == 0)
            x = 10;
    else
        x = -1;
    cout << x << endl;
}
```

- a) 1.
- b) 5.
- c) 0.
- d) -1.

3) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

int main( )
{
    using namespace std;
    int x = 1;

    switch (x)
    {
        case 1:
            x++;
        case 2:
            x++;
    }
    cout << x << endl;
}
```

- a) 1.
- b) 2.
- c) 3.
- d) Ninguno de los anteriores.

4) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
int main( )
{
```



```
using namespace std;
int x = 1;

while (x <= 5)
{
    cout << ++x << ' ';
}
}
```

- a) 1 2 3 4 0.
- b) 2 3 4 5 6.
- c) 0 1 2 3 4.
- d) Ninguno de los anteriores.

5) ¿Cuál es el resultado del siguiente programa? (El valor ASCII de 'a' es 97).

```
#include <iostream>

int main( )
{
    using namespace std;
    int x = 0;

    for (x = 'a'; x <= 'z'; x += 10)
    {
        cout << x << ' ';
    }
}
```

- a) a b c d e ... x y z.
- b) 97 98 99 ... 120 121 122.
- c) 97 107 117.
- d) a k u.

6) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

int main( )
{
    using namespace std;
    int x = 0, y = 0;

    for (x = 6; x > 0; x -= 2)
        for (y = 0; y < 2; y++)
            cout << x-y << ' ';
}
```

- a) 6 5 4 3 2 1.
- b) 6 4 2.
- c) 6 2.
- d) Ninguno de los anteriores.

7) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

int main( )
{
    using namespace std;
    int x = 0, y = 1;

    for (x = 6; x > 0; x -= 2)
    {
        if (x < y * 3) continue;
        cout << x << ' ';
    }
}
```

- a) 6 4 2.
- b) 6 4.
- c) 6.
- d) Ninguno de los anteriores.

8) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

void fnx(int x)
{
    using namespace std;
    if (x) cout << x << ' ';
}

int main( )
{
    int i, a = 1234;
    for (i = 0; i < 4; i++)
        fnx(a = a/10);
}
```

- a) 123 12 1.
- b) 1 2 3 4.
- c) 4 3 2 1.
- d) Ninguno de los anteriores.

9) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

void fnx(int x)
{
    int i = 0;
    for (i = x; i > 0; i--)
        std::cout << i << ' ';
}

int main( )
{
    int x;

    for (x = 0; x < 3; x++)
        fnx(x);
}
```

- a) 0.
- b) 1 2 1.
- c) 1 2.
- d) Ninguno de los anteriores.

10) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

void fnx(int x)
{
    std::cout << static_cast<char>(x) << ' ';
}

int main( )
{
    int i, x = 65;
    for (i = 0; i < 3; i++)
        fnx(x++);
}
```

- a) A B C.
- b) 65 66 67.
- c) a b c.
- d) Ninguno de los anteriores.

2. Realizar un programa que calcule e imprima la suma de los múltiplos de 5 comprendidos entre dos valores a y b . El programa no permitirá introducir valores negativos para a y b , y verificará que a es menor que b . Si a es mayor que b , intercambiará estos valores.

3. Si quiere averiguar su número de Tarot, sume los números de su fecha de nacimiento y a continuación redúzcalos a un único dígito; por ejemplo si su fecha de nacimiento fuera 26 de octubre de 1983, los cálculos a realizar serían:

$$26 + 10 + 1983 = 2019 \Rightarrow 2 + 0 + 1 + 9 = 12 \Rightarrow 1 + 2 = 3$$

lo que quiere decir que su número de Tarot es el 3.

Realizar un programa que pida una fecha, de la forma:

día mes año

donde *día*, *mes* y *año* son enteros, y dé como resultado el número de Tarot. El programa verificará si la fecha es correcta, esto es, los valores están dentro de los rangos permitidos.

4. Un centro numérico es un número que separa una lista de números enteros (comenzando en 1) en dos grupos de números, cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista (1 a 8) en los grupos: (1, 2, 3, 4, 5) y (7, 8) cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista (1 a 49) en los grupos: (1 a 34) y (36 a 49) cuyas sumas son ambas iguales a 595. Escribir un programa que calcule los centros numéricos entre 1 y n .
5. Escribir un programa para que lea un texto y dé como resultado el número de palabras con al menos cuatro vocales diferentes. Suponemos que una palabra está separada de otra por uno o más espacios (' '), tabuladores (\t) o caracteres '\n'. La entrada de datos finalizará cuando se detecte la marca de fin de fichero. La ejecución será de la forma siguiente:

```
Introducir texto. Para finalizar introducir la marca eof.
En la Universidad hay muchos
estudiantes de Telecomunicación
[Ctrl] [z]
```

Número de palabras con 4 vocales distintas: 3

6. Escribir un programa para que lea un texto y dé como resultado el número de caracteres, palabras y líneas del mismo. Suponemos que una palabra está separada de otra por uno o más espacios (' '), caracteres *tab* (\t) o caracteres '\n'. La ejecución será de la forma siguiente:

```
Introducir texto. Pulse [Entrar] después de cada línea.
Para finalizar pulsar Ctrl+z.
```

Este programa cuenta los caracteres, las palabras y las líneas de un documento.

```
[Ctrl] [z]
```

```
80 13 2
```

7. Escribir un programa que calcule la serie:

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

Para un valor de x dado, se calcularán y sumarán términos sucesivos de la serie, hasta que el último término sumado sea menor o igual que una constante de error predeterminada (por ejemplo $1e-7$). Observe que cada término es igual al anterior por x/n para $n = 1, 2, 3, \dots$. El primer término es el 1. Para ello se pide:

- a) Escribir una función que tenga el siguiente prototipo:

```
double exponencial(double x);
```

Esta función devolverá como resultado el valor aproximado de e^x .

- b) Escribir la función **main** para que invoque a la función *exponencial* y compruebe que para x igual a 1 el resultado es el número e .

TIPOS ESTRUCTURADOS DE DATOS

Hasta ahora sólo hemos tenido que trabajar con algunas variables en cada uno de los programas que hemos realizado. Sin embargo, en más de una ocasión tendremos que manipular conjuntos más grandes de valores. Por ejemplo, para calcular la temperatura media del mes de agosto necesitaremos conocer los 31 valores correspondientes a la temperatura media de cada día. En este caso, podríamos utilizar una variable para introducir los 31 valores, uno cada vez, y acumular la suma en otra variable. Pero, ¿qué ocurrirá con los valores que vayamos introduciendo? Que cuando tecleemos el segundo valor, el primero se perderá; cuando tecleemos el tercero, el segundo se perderá, y así sucesivamente. Cuando hayamos introducido todos los valores podremos calcular la media, pero las temperaturas correspondientes a cada día se habrán perdido. ¿Qué podríamos hacer para almacenar todos esos valores? Pues podríamos utilizar 31 variables diferentes; pero, ¿qué pasaría si fueran 100 o más valores los que tuviéramos que registrar? Además de ser muy laborioso el definir cada una de las variables, el código se vería enormemente incrementado.

En este capítulo, aprenderá a registrar conjuntos de valores, todos del mismo tipo, en unas estructuras de datos llamadas *matrices*. Así mismo, aprenderá a registrar cadenas de caracteres, que no son más que conjuntos de caracteres, o bien, si lo prefiere, matrices de caracteres.

Si las matrices son la forma de registrar conjuntos de valores, todos del mismo tipo (**int**, **float**, **double**, **char**, etc.), ¿qué haremos para almacenar un conjunto de valores relacionados entre sí, pero de diferentes tipos? Por ejemplo, almacenar los datos relativos a una persona como su *nombre*, *dirección*, *teléfono*, etc. Veremos que esto se hace definiendo otro tipo de estructura de datos; en este caso,

podría ser del tipo `struct persona`. También podremos crear matrices de este tipo de estructuras, cuestión que aprenderemos más adelante.

INTRODUCCIÓN A LAS MATRICES

Una matriz es una estructura homogénea, compuesta por varios elementos, todos del mismo tipo y almacenados consecutivamente en memoria. Cada elemento puede ser accedido directamente por el nombre de la variable matriz seguido de uno o más subíndices encerrados entre corchetes.



En general, la representación de las matrices se hace mediante variables suscritas o de subíndices y pueden tener una o varias dimensiones (subíndices). A las matrices de una dimensión se les llama también listas y a las de dos dimensiones, tablas.

Desde un punto de vista matemático, en más de una ocasión necesitaremos utilizar variables subindicadas tales como:

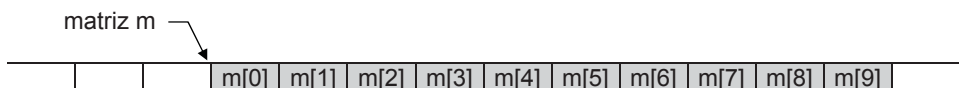
$$v = [a_0, a_1, a_2, \dots, a_i, \dots, a_n]$$

en el caso de un subíndice, o bien

$$m = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0j} & \dots & a_{0n} \\ a_{10} & a_{11} & a_{12} & \dots & a_{1j} & \dots & a_{1n} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i0} & a_{i1} & a_{i2} & \dots & a_{ij} & \dots & a_{in} \end{pmatrix}$$

si se utilizan dos subíndices. Esta misma representación se puede utilizar desde un lenguaje de programación recurriendo a las matrices que acabamos de definir y que a continuación se estudian.

Por ejemplo, supongamos que tenemos una matriz unidimensional de enteros llamada m , la cual contiene 10 elementos. Estos elementos se identificarán de la siguiente forma:



Observe que los subíndices son enteros consecutivos y que el primer subíndice vale 0. Un subíndice puede ser cualquier expresión entera positiva.

Así mismo, una matriz de dos dimensiones se representa mediante una variable con dos subíndices (filas, columnas); una matriz de tres dimensiones se representa mediante una variable con tres subíndices, etc. El número máximo de dimensiones o el número máximo de elementos para una matriz, dentro de los límites establecidos por el compilador, depende de la memoria disponible.

Entonces, las matrices según su dimensión se clasifican en unidimensionales y multidimensionales; y según su contenido, en numéricas, de caracteres, de estructuras, de punteros y de objetos. A continuación se estudia todo esto detalladamente.

MATRICES NUMÉRICAS UNIDIMENSIONALES

Igual que sucede con otras variables, antes de utilizar una matriz hay que definirla. La definición de una matriz especifica el nombre de la matriz, el número de elementos de la misma y el tipo de éstos.

Definir una matriz

La definición de una matriz de una dimensión se hace de la forma siguiente:

```
tipo nombre[tamaño];
```

donde *tipo* indica el tipo de los elementos de la matriz, el cual puede ser cualquier tipo primitivo o definido por el usuario; *nombre* es un identificador que nombra a la matriz y *tamaño* es una constante entera que especifica el número de elementos de la matriz. Los corchetes modifican la definición normal del identificador para que sea interpretado por el compilador como una matriz.

Veamos algunos ejemplos. Las siguientes líneas de código son ejemplos de definiciones de matrices:

```
int m[10];
float temperatura[31];
COrdenador ordenador[25];
```

La primera línea crea una matriz identificada por *m* con 10 elementos de tipo **int**; es decir, puede almacenar 10 valores enteros; el primer elemento es *m[0]* (se lee: *m sub-cero*), el segundo *m[1]*, ..., y el último *m[9]*. La segunda crea una ma-

triz *temperatura* de 31 elementos de tipo **float**. Y la tercera crea una matriz *ordenador* de 25 elementos, cada uno de los cuales es un objeto de tipo *COrdenador*.

Acceder a los elementos de una matriz

Para acceder al valor de un elemento de una matriz se utiliza el nombre de la matriz, seguido de un subíndice entre corchetes. Esto es, un elemento de una matriz no es más que una variable subindicada; por lo tanto, se puede utilizar exactamente igual que cualquier otra variable. Por ejemplo, en las operaciones que se muestran a continuación intervienen elementos de una matriz:

```
int m[100], k = 0, a = 0;
// ...
a = m[1] + m[99];
k = 50;
m[k]++;
m[k+1] = m[k];
```

Observe que para referenciar un elemento de una matriz se puede emplear como subíndice una constante, una variable o una expresión de tipo entero. El subíndice especifica la posición del elemento dentro de la matriz. La primera posición es la 0.

Si se intenta acceder a un elemento con un subíndice menor que 0 o mayor que el número de elementos de la matriz menos 1, C++ no informa de ello porque no realiza ese tipo de chequeo; es el sistema operativo el que lo notificará mediante un mensaje de error, sólo si ese intento transgrede los límites de la zona de memoria asignada por él a dicha aplicación. Por lo tanto, es responsabilidad del programador escribir el código necesario para detectar este tipo de error. Por ejemplo, la última línea de código del ejemplo siguiente dará lugar a un resultado impredecible, puesto que intenta asignar el valor del elemento de subíndice 99 al elemento de subíndice 100, que está fuera del rango 0 a 99 válido.

```
int m[100], k = 0, a = 0;
// ...
k = 99;
m[k+1] = m[k];
```

¿Cómo podemos asegurarnos de no exceder accidentalmente los límites de una matriz? Verificando en todo momento que el índice está entre 0 y la longitud de la matriz menos 1. Por ejemplo:

```
const int N = 100;
// ...
```

```

int m[N], k = 0, a = 0;
// ...
k = 99;
if (k >= 0 && k < N-1)
    m[k+1] = m[k];
else
    cout << "índice fuera de límites\n";

```

Iniciar una matriz

Cuando durante la ejecución de un programa ocurre la definición de una matriz, sus elementos son automáticamente iniciados sólo si la definición se ha realizado a nivel global; en este caso, igual que sucedía con las variables globales, si la matriz es numérica, sus elementos son iniciados a 0; si es de caracteres, al valor ‘\0’ y si es de punteros, a 0. Cuando la matriz sea local sus elementos no serán iniciados automáticamente; en este caso, ¿qué valores almacenarán? Valores indeterminados; dicho de otra forma, almacenarán basura. Si la matriz es local pero se declara **static**, entonces se inicia igual que si fuera global.

Ahora bien, si deseamos iniciar una matriz con unos valores determinados en el momento de definirla, podemos hacerlo de la siguiente forma:

```
float temperatura[6] = {10.2F, 12.3F, 3.4F, 14.5F, 15.6F, 16.7F};
```

En el ejemplo anterior, el tamaño de la matriz debe ser igual o mayor que el número de valores especificados. Cuando es mayor, sólo serán iniciados explícitamente tantos elementos como valores. El resto de los elementos serán iniciados implícitamente, dependiendo del tipo, a 0, a ‘\0’ o a **NULL** (dirección 0).

Y si deseamos iniciar todos los elementos de una matriz a 0 en el momento de definirla, podemos hacerlo de la siguiente forma:

```
float temperatura[6] = {0};
```

Siempre que se inicie una matriz en el instante de su definición, el tamaño puede omitirse; en este caso el número de elementos se corresponderá con el número de valores especificados. También puede omitirse el tamaño en los siguientes casos: cuando se declara como un parámetro formal en una función y cuando se hace referencia a una matriz declarada en otra parte del programa.

Por ejemplo, en el siguiente programa se puede observar que la función **main** define e inicia una matriz *x* sin especificar explícitamente su tamaño y que el primer parámetro de la función *VisualizarMatriz* es una matriz *m* de la cual tampoco se especifica su tamaño. En este último caso, ¿cuál es el tamaño de la matriz *m*?

Lógicamente el tamaño de la matriz x pasada como argumento a la función *VisualizarMatriz*.

```
#include <iostream>
using namespace std;
void VisualizarMatriz(int [], int);

int main( )
{
    int x[] = { 10, 20, 30, 40, 50 };
    VisualizarMatriz(x, 5);
}

void VisualizarMatriz(int m[], int n)
{
    int i = 0;
    for (i = 0; i < n; i++)
        cout << m[i] << ' ';
}
```

Así mismo, en el ejemplo anterior se puede observar que cuando **main** invoca a *VisualizarMatriz* le pasa dos argumentos: la matriz x y el número de elementos. Por lo tanto, el argumento x es copiado en el parámetro m y el argumento 5 en n . Dos preguntas: ¿qué se ha copiado en m ? ¿Se ha hecho un duplicado de la matriz? No, no se ha hecho un duplicado. Cuando el argumento es una matriz lo que se pasa es la dirección de esa matriz; esto es, la posición donde comienza el bloque de memoria que ocupa físicamente esa matriz. Dicha dirección viene dada por el nombre de la matriz. Esto quiere decir que C++ siempre pasa las matrices por referencia. Según lo expuesto, ¿qué conoce *VisualizarMatriz* de la matriz x ? Pues la dirección de comienzo, m , y su número de elementos, n ; entonces $m[i]$ en esta función y $x[i]$ en **main**, ¿son el mismo elemento? Sí, porque las dos funciones trabajan a partir de la misma posición de memoria y sobre el mismo número de elementos. Quiere esto decir que si *VisualizarMatriz* modificara la matriz, esas modificaciones afectarían también a **main**. Como ejercicio puede comprobarlo.

Trabajar con matrices unidimensionales

Para practicar la teoría expuesta hasta ahora, vamos a realizar un programa que asigne datos a una matriz unidimensional m de $N_ELEMENTOS$ y, a continuación, como comprobación del trabajo realizado, escriba el contenido de dicha matriz. La solución será similar a la siguiente:

```
La matriz tiene 10 elementos.
Introducir los valores de la matriz.
m[0] = 12
m[1] = 5
```

```
m[2] = -75
...
12 5 -75...
```

Fin del proceso.

Para ello, en primer lugar definimos la constante *N_ELEMENTOS* para fijar el número de elementos de la matriz:

```
const int N_ELEMENTOS = 10;
```

Después creamos la matriz *m* con ese número de elementos y definimos el subíndice *i* para acceder a los elementos de dicha matriz.

```
int m[N_ELEMENTOS]; // crear la matriz m
int i = 0; // subíndice
```

El paso siguiente es asignar un valor desde el teclado a cada elemento de la matriz.

```
for (i = 0; i < N_ELEMENTOS; i++)
{
    cout << "m[" << i << "] = ";
    cin >> m[i];
}
```

Una vez leída la matriz la visualizamos para comprobar el trabajo realizado.

```
for (i = 0; i < N_ELEMENTOS; i++)
    cout << m[i] << ' ';
```

El programa completo se muestra a continuación:

```
// Creación de una matriz unidimensional
#include <iostream>
using namespace std;

int main()
{
    const int N_ELEMENTOS = 10;
    int m[N_ELEMENTOS]; // crear la matriz m
    int i = 0;          // subíndice

    cout << "Introducir los valores de la matriz.\n";
    // Entrada de datos
    for (i = 0; i < N_ELEMENTOS; i++)
    {
```

```

        cout << "m[" << i << "] = ";
        cin >> m[i];
    }

    // Salida de datos
    cout << "\n\n";
    for (i = 0; i < N_ELEMENTOS; i++)
        cout << m[i] << ' ';
    cout << "\n\nFin del proceso.\n";
}

```

Tipo y tamaño de una matriz

En el capítulo 4 vimos que utilizando **typedef** podíamos declarar sinónimos de otros tipos fundamentales o derivados. El tipo matriz es un tipo derivado (por ejemplo, **int []** para una matriz entera de una dimensión). Entonces, se puede declarar un sinónimo de un tipo matriz así:

```
typedef double t_matriz_1d[100];
```

La línea anterior define un nuevo tipo, *t_matriz_1d*, que define matrices unidimensionales de 100 elementos de tipo **double**. Y la siguiente sentencia utiliza este tipo para definir una matriz *m*:

```
t_matriz_1d m;
```

Así mismo, vimos que el operador **sizeof** daba como resultado el tamaño en bytes de su operando. Pues bien, cuando el operando es una matriz, el resultado es el tamaño en bytes de dicha matriz. Por ejemplo, la siguiente línea de código visualiza el número de elementos de la matriz *m* definida anteriormente:

```
cout << "Nº de elementos: " << sizeof(m)/sizeof(m[0]) << endl;
```

o bien, podemos escribir también:

```
cout << "Nº de elementos: " << sizeof(m)/sizeof(double) << endl;
```

Lo expuesto aquí puede hacerse extensivo a las matrices multidimensionales que explicaremos un poco más adelante.

Vector

La biblioteca estándar de C++ proporciona una plantilla **vector** definida en el espacio de nombres **std** y declarada en el fichero de cabecera `<vector>` que facilita

la creación y manipulación de matrices. Por ejemplo, para definir una matriz de una dimensión, la sintaxis a utilizar es la siguiente:

```
vector<tipo> nombre([tamaño, [val]]);
```

donde *tipo* indica el tipo de los elementos de la matriz, el cual puede ser cualquier tipo primitivo o definido por el usuario; *nombre* es un identificador que nombra a la matriz (objeto matriz); *tamaño* es una variable entera que especifica el número de elementos de la matriz y *val* es un parámetro opcional, del mismo tipo que los elementos, cuyo valor será utilizado para iniciar los elementos de la matriz.

Por ejemplo, la matriz *m* del ejemplo anterior podría declararse también así:

```
vector<int> m(N_ELEMENTOS);
```

En este caso, al tratarse de un tipo primitivo, los elementos de la matriz son automáticamente iniciados a cero, cosa que no ocurría antes.

Esta forma de declarar una matriz no resulta más complicada y favorece en el sentido de que el número de elementos no necesita ser constante. Por ejemplo:

```
int n_elementos;
cout << "Número de elementos: "; cin >> n_elementos;
vector<int> m(n_elementos, 0); // crear la matriz m iniciada a 0
```

También un vector puede ser copiado en una única operación en otro vector, operación que no puede realizarse con las matrices primitivas. Por ejemplo:

```
vector<int> v; // vector v con ceros elementos.
v = m; // copiar el vector m en v.
// v es redimensionado al tamaño de m.
```

Naturalmente, la plantilla **vector** proporciona métodos para acceder a los elementos del vector, insertar nuevos elementos, eliminar elementos, obtener el número de elementos, asignar un nuevo tamaño, etc. A continuación veremos algunos de los métodos más comunes. Evidentemente esto marca una clara diferencia frente a las matrices primitivas de C++, por lo que es aconsejable su utilización.

Acceso a los elementos

Para acceder a un elemento podemos utilizar el operador `[]` o el método `at`. De la primera manera no disponemos de verificación de si el índice está fuera del rango y de la segunda sí (ver más adelante, en otro capítulo, *Excepciones*).

```
vector<int> v(20); // vector v con 20 elementos
```

```
v[i]++;           // sin verificación de rango
v.at(i)++;       // con verificación de rango
```

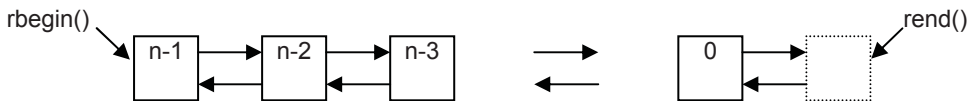
Los métodos **front/back** permiten acceder al primer y último elemento, respectivamente. Como veremos a continuación, también es posible acceder a un elemento a través de un iterador.

Iteradores

Los iteradores se utilizan para navegar a través de los contenedores sin necesidad de conocer el tipo utilizado para identificar los elementos. Los iteradores **begin/end** ofrecen los elementos en el orden normal (0, 1, 2, ..., n-2, n-1); **begin** apunta al primer elemento de la secuencia y **end** al elemento siguiente al último.



Los iteradores **rbegin/rend** ofrecen los elementos en el orden inverso (n-1, n-2, ..., 1, 0); **rbegin** apunta al primer elemento de la secuencia inversa y **rend** al elemento siguiente al último.



```
vector<int> v(20);           // vector v con 20 elementos

vector<int>::iterator e;    // e es un iterador

int k = 0;
for (e = v.begin(); e != v.end(); e++)
    *e = k++; // asignar al elemento referenciado por e el valor k

// ...
e = v.begin()+2;
k = *e; // obtener el valor del elemento referenciado por e
```

Obsérvese que **e* es el entero referenciado por *e*. Esto quiere decir que un iterador define un puntero. Según esto, podríamos escribir también:

```
for (int *p = &v[0]; p < &v[v.size()]; p++)
    *p = k++; // asignar al elemento apuntado por p el valor k
```


Tamaño

Para conocer el tamaño de un vector hay que invocar a su método **size** y para modificar su tamaño, al método **resize**; en este último caso, los elementos conservados desde el primero hasta el nuevo tamaño permanecen inalterados.

```
int nElementos = v.size(); // tamaño
v.resize(nElementos - 2); // nuevo tamaño
```

Para saber si un vector está vacío, utilizar el método **empty**. Este método devuelve **true** si el vector está vacío y **false** en caso contrario.

Eliminar elementos

El método **pop_back** permite eliminar el elemento del final y el método **erase** permite eliminar desde un elemento hasta otro.

```
v.pop_back(); // elimina el último elemento
e = v.begin(); // referencia al primer elemento
v.erase(e+3, e+5); // eliminar v[3] y v[4]
```

Para eliminar todos los elementos utilizar el método **clear**.

Buscar elementos

El algoritmo **find** declarado en *<algorithm>* permite buscar un elemento en un contenedor; por ejemplo, en un **vector**. Esta función devuelve un iterador al primer elemento que coincide con el valor. Por ejemplo:

```
vector<int> v(10);
vector<int>::iterator e; // e es un iterador
// ...
e = find(v.begin(), v.end(), n);
if (*e == n)
{
    v.erase(e);
    cout << "elemento eliminado\n";
}
else
    cout << "error: elemento no encontrado\n";
```

Insertar elementos

El método **push_back** permite añadir un elemento al final y el método **insert** permite insertar uno o más elementos en cualquier posición.

```

k = 17;
v.push_back(k);
v.insert(v.begin()+3, 2, -1); // insertar 2 elementos desde v[3]
                             // (v[3] y v[4]) con valor -1

```

Los métodos **push_back** y **pop_back** permiten utilizar un vector como una pila (las pilas serán estudiadas en el capítulo *Estructuras dinámicas*).

```

vector<int> pila; // vector con 0 elementos
pila.push_back(1);
pila.push_back(2);
pila.push_back(3);
cout << pila.back() << endl; // escribe 3
pila.pop_back();
cout << pila.back() << endl; // escribe 2

```

Ejemplo

El ejercicio anterior nos enseña cómo leer una matriz y cómo escribirla. El paso siguiente es aprender a trabajar con los valores almacenados en la matriz. En este caso utilizaremos la plantilla **vector**. Por ejemplo, pensemos en un programa que lea la nota media obtenida por cada alumno de un determinado curso, las almacene en una matriz y dé como resultado la nota media del curso.

Igual que hicimos en el programa anterior, en primer lugar crearemos una matriz *notas* con un número de elementos cero inicialmente. En este caso interesa que la matriz sea de tipo **float** para que sus elementos puedan almacenar un valor con decimales.

```

vector<float> notas; // matriz notas inicialmente vacía

```

Para garantizar una entrada correcta de un valor numérico, reutilice los ficheros *utils.h* y *utils.cpp* creados en el capítulo *Entrada y salida estándar*.

El paso siguiente será almacenar en la matriz las notas introducidas a través del teclado. La entrada finalizará tecleando la marca de fin de fichero (*Ctrl+z*).

```

float nota = LeerFloat();
while (nota != -1)
{
    notas.push_back(nota); // añadir un elemento
    cout << "Alumno número" << setw(3) << ++i << ", nota media: ";
    nota = LeerFloat();
}

```

Finalmente se suman todas las notas y se visualiza la nota media. La suma se almacenará en la variable *suma*. Una variable utilizada de esta forma recibe el nombre de acumulador. Es importante que observe que inicialmente su valor es 0.

```
int nalumnos = notas.size(); // número de alumnos
float suma = 0; // suma total de todas las notas medias
for (i = 0; i < nalumnos; i++)
    suma += notas[i];
cout << "\n\nNota media del curso: "
     << fixed << setprecision(2) << setw(5)
     << suma / nalumnos << endl;
```

El programa completo se muestra a continuación.

```
// Nota media del curso
#include <iostream>
#include <iomanip>
#include <limits>
#include <vector>
#include "utils.h" // para LeerFloat()
using namespace std;

int main( )
{
    vector<float> notas; // matriz notas inicialmente vacía

    // Entrada de datos
    int i = 0;
    cout << "Introducir notas. Finalizar con -1\n";
    cout << "Alumno número" << setw(3) << ++i << ", nota media: ";
    float nota = LeerFloat();
    while (nota != -1)
    {
        notas.push_back(nota); // añadir un elemento
        cout << "Alumno número" << setw(3) << ++i << ", nota media: ";
        nota = LeerFloat();
    }
    cin.clear(); // desactivar los indicadores de error

    // Sumar las notas
    int nalumnos = notas.size(); // número de alumnos
    float suma = 0; // suma total de todas las notas medias
    for (i = 0; i < nalumnos; i++)
        suma += notas[i];
    // Escribir resultados
    cout << "\n\nNota media del curso: "
         << fixed << setprecision(2) << setw(5)
         << suma / nalumnos << endl;
}
```

Ejecución del programa

```

Alumno número 1, nota media: 5.5
Alumno número 2, nota media: 9
Alumno número 3, nota media: 7.5
Alumno número 4, nota media: 6
Alumno número 5, nota media: 8
Alumno número 6, nota media: ^Z

```

```
Nota media del curso: 7.20
```

Matrices asociativas

En una matriz asociativa el acceso a los elementos se hace por valor en lugar de por posición (por ejemplo, una matriz *diasMes* que almacene en el elemento de índice 1 los días del mes 1, en el de índice 2 los días del mes 2 y así sucesivamente; ignoramos el elemento de índice 0). En estos casos, la solución del problema resultará más fácil si utilizamos esa coincidencia. Por ejemplo, vamos a realizar un programa que cuente el número de veces que aparece cada una de las letras de un texto introducido por el teclado y a continuación imprima el resultado. Para hacer el ejemplo sencillo, vamos a suponer que el texto sólo contiene letras minúsculas del alfabeto inglés (no hay ni letras acentuadas, ni la *ll*, ni la *ñ*). La solución podría ser de la forma siguiente:

```

Introducir texto (para finalizar introducir la marca eof):
las matrices mas utilizadas son las unidimensionales
y las bidimensionales.

```

```
[Ctrl][z]
```

```

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 z
-----
9 1 1 3 5 0 0 0 9 0 0 6 4 6 3 0 0 1 1 1 2 2 0 0 0 1 1

```

Antes de empezar el problema, vamos a analizar algunas de las operaciones que después utilizaremos en el programa. Por ejemplo, la expresión:

```
'z' - 'a' + 1
```

da como resultado 26. Recuerde que cada carácter tiene asociado un valor entero (código ASCII) que es el que utiliza la máquina internamente para manipularlo. Así, por ejemplo, la 'z' tiene asociado el entero 122, la 'a' el 97, etc. Según esto, la evaluación de la expresión anterior es: $122 - 97 + 1 = 26$.

Por la misma razón, si realizamos las declaraciones,

```
int c[256]; // la tabla ASCII tiene 256 caracteres
```

```
char car = 'a'; // car tiene asignado el entero 97
```

la siguiente sentencia asigna a $c[97]$ el valor 10,

```
c['a'] = 10;
```

y esta otra sentencia que se muestra a continuación realiza la misma operación, lógicamente, suponiendo que *car* tiene asignado el carácter 'a'.

```
c[car] = 10;
```

Entonces, si leemos un carácter (de la 'a' a la 'z'),

```
cin.get(car);
```

y a continuación realizamos la operación,

```
c[car]++;
```

¿qué elemento de la matriz *c* se ha incrementado? La respuesta es el de subíndice igual al código correspondiente al carácter leído. Hemos hecho coincidir el carácter leído con el subíndice de la matriz. Así, cada vez que leamos una 'a' se incrementará el contador $c[97]$ o lo que es lo mismo $c['a']$; tenemos entonces un contador de 'a'. Análogamente diremos para el resto de los caracteres.

Pero, ¿qué pasa con los elementos $c[0]$ a $c[96]$? Según hemos planteado el problema inicial quedarían sin utilizar (el enunciado decía: con qué frecuencia aparecen los caracteres de la 'a' a la 'z'). Esto, aunque no presenta ningún problema, se puede evitar así:

```
c[car - 'a']++;
```

Para *car* igual a 'a' se trataría del elemento $c[0]$ y para *car* igual a 'z' se trataría del elemento $c[25]$. De esta forma podemos definir una matriz de enteros justamente con un número de elementos igual al número de caracteres de la 'a' a la 'z' (26 caracteres según la tabla ASCII). El primer elemento será el contador de 'a', el segundo el de 'b', y así sucesivamente.

Un contador es una variable que inicialmente vale 0 (suponiendo que la cuenta empieza desde 1) y que después se incrementa en una unidad cada vez que ocurre el suceso que se desea contar.

El programa completo se muestra a continuación.

```
// Frecuencia con la que aparecen las letras en un texto
```

```

#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

int main( )
{
    const int N_ELEMENTOS = 'z'-'a'+1; // número de elementos
    vector<int> c(N_ELEMENTOS); // matriz c; sus elementos están a 0
    char car; // índice

    // Entrada de datos y cálculo de la tabla de frecuencias
    cout << "Introducir texto. ";
    cout << "(para finalizar introducir la marca eof):\n\n";
    while (cin.get(car), !cin.eof())
    {
        // Si el carácter leído está entre la 'a' y la 'z'
        // incrementar el contador correspondiente
        if (car >= 'a' && car <= 'z')
            c[car - 'a']++;
    }
    // Escribir la tabla de frecuencias
    for (car = 'a'; car <= 'z'; car++)
        cout << " " << car;
    cout << "\n -----"
         << "\n";

    for (car = 'a'; car <= 'z'; car++)
        cout << setw(3) << c[car - 'a'];
    cout << endl;
}

```

Map

La biblioteca estándar de C++ proporciona una plantilla **map** definida en el espacio de nombres **std** y declarada en el fichero de cabecera `<map>` que facilita la creación y manipulación de matrices asociativas. Para definir una de estas matrices, la sintaxis a utilizar es la siguiente:

```
map<tipo1, tipo2> nombre;
```

donde se observa que un mapa (también conocido como matriz asociativa o diccionario) es un contenedor de pares de valores: el primer elemento del par, de *tipo1*, es la clave, que se utiliza como índice para buscar el segundo elemento del par, de *tipo2*, correspondiente al valor asociado; y *nombre* es un identificador que nombra al mapa.

Es frecuente emplear como clave un dato de tipo **string** y como valor asociado cualquier dato de un tipo primitivo o definido por el usuario.

Según lo expuesto, la matriz *c* del ejemplo anterior podría ser sustituida por:

```
map<char, int> c;
```

En este caso, la matriz asociativa *c* está inicialmente vacía. Para añadir un elemento (un par de valores) basta con hacer referencia al mismo y será añadido siempre y cuando no exista. Por ejemplo:

```
map<char, int> c; // matriz asociativa c inicialmente vacía
int x = c['a']; // crea nueva entrada para 'a' iniciada a 0 (x = 0)
c['b'] = 5; // crea nueva entrada para 'b' y le asigna 5
c['a'] = 7; // cambia el valor de la entrada de 'a' a 7
int y = c['b']; // devuelve el valor de la entrada de 'b' (y = 5)
```

De acuerdo con esto, podemos volver a escribir el ejemplo expuesto anteriormente como se indica a continuación:

```
// Frecuencia de las letras en un texto
#include <iostream>
#include <iomanip>
#include <map>

using namespace std;

int main( )
{
    map<char, int> c; // matriz asociativa c inicialmente vacía
    char car; // índice

    // Entrada de datos y cálculo de la tabla de frecuencias
    cout << "Introducir texto. ";
    cout << "(para finalizar introducir la marca eof):\n\n";
    while (cin.get(car), !cin.eof())
    {
        // Si el carácter leído está entre la 'a' y la 'z'
        // incrementar el contador correspondiente
        if (car >= 'a' && car <= 'z')
            c[car]++;
    }

    // Escribir la tabla de frecuencias
    for (car = 'a'; car <= 'z'; car++)
        cout << " " << car;
    cout << "\n -----"
         << "\n";
```

```

for (car = 'a'; car <= 'z'; car++)
    cout << setw(3) << c[car];
cout << endl;
}

```

La plantilla **map** proporciona también los atributos **first** y **second** para acceder al par de valores de un elemento, iteradores, el método **find** para buscar un elemento, el método **erase** para eliminar un rango de elementos, el método **clear** para eliminar todos los elementos, el método **size** para obtener el número de elementos, el método **empty** para saber si hay elementos, etc. A continuación vemos algunos ejemplos:

```

// Operaciones con mapas
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main( )
{
    map<string, long> agenda; // agenda inicialmente vacía
    string nombre;          // índice
    long telefono = 0;
    map<string, long>::iterator persona;
    // Añadir elementos a la agenda
    cout << "Introducir pares de valores nombre teléfono "
         << "(finalizar con eof):\n";
    while (cin >> nombre >> telefono) agenda[nombre] = telefono;
    // Borrar el elemento identificado por "xxxx"
    if (!agenda.empty())
    {
        persona = agenda.find("xxxx");
        if (persona != agenda.end())
        {
            cout << persona->first << " borrado\n";
            agenda.erase(persona);
        }
    }
    // Mostrar los elementos de la agenda
    for (persona = agenda.begin(); persona != agenda.end(); persona++)
        cout << " " << (*persona).first << '\t'
             << (*persona).second << endl;
    // Obtener el número de elementos
    cout << agenda.size() << "elementos\n";
    // Borrar todos los elementos
    agenda.clear();
}

```


CADENAS DE CARACTERES

En los capítulos anteriores ya hemos hablado en más de una ocasión de las cadenas de caracteres. Por ejemplo, la línea de código siguiente visualiza el literal “*Fin del proceso.*”. Dicho literal es una cadena de caracteres constante.

```
cout << "Fin del proceso.\n";
```

Básicamente, una cadena de caracteres se almacena como una matriz unidimensional de elementos de tipo **unsigned char** o **char**:

```
char cadena[10];
```

Igual que sucedía con las matrices numéricas, una matriz unidimensional de caracteres puede ser iniciada en el momento de su definición. Por ejemplo:

```
char cadena[] = {'a', 'b', 'c', 'd', '\0'};
```

Este ejemplo define *cadena* como una matriz de caracteres con cinco elementos (*cadena[0]* a *cadena[4]*) y asigna al primer elemento el carácter ‘a’, al segundo el carácter ‘b’, al tercero el carácter ‘c’, al cuarto el carácter ‘d’ y al quinto el carácter nulo.

Puesto que cada carácter es un entero, el ejemplo anterior podría escribirse también así:

```
char cadena[] = {97, 98, 99, 100, 0};
```

Cada carácter tiene asociado un entero entre 0 y 255 (código ASCII). Por ejemplo, a la ‘a’ le corresponde el valor 97, a la ‘b’ el valor 98, etc. Entonces, una cadena de caracteres no es más que una matriz de enteros.

Si se crea una matriz de caracteres y se le asigna un número de caracteres menor que su tamaño, el resto de los elementos quedan con el valor ‘\0’ independientemente de que la matriz sea global o local. Por ejemplo:

```
char cadena[40] = "abcd";
```

Para visualizar una cadena invocaremos al operador << sobre el objeto **cout**. Se visualizarán todos los caracteres de la cadena hasta encontrar un carácter nulo (‘\0’). Las funciones de la biblioteca de C++ como **getline**, que veremos a continuación, finalizan automáticamente las cadenas leídas con un carácter ‘\0’. En los casos en los que esto no suceda, ¿qué ocurriría? Veamos un ejemplo:

```
char cadena[10];
```

```
cadena[0] = 'a'; cadena[1] = 'b'; cadena[2] = 'c'; cadena[3] = 'd';
cadena[4] = '\0'; // asignar el valor ASCII 0
printf("%s\n", cadena);
```

En el ejemplo anterior, suponiendo que *cadena* sea una variable local, si no asignamos explícitamente a *cadena[4]* un carácter ‘\0’, lo más seguro es que **cout** escriba *abcd* seguido de basura. Incluso podría escribir más allá del límite de la cadena, lo que de por sí ya es una operación errónea.

El proceso de iniciación descrito anteriormente lo tiene automatizado C++ si se inicia la cadena como se indica a continuación:

```
char cadena[10] = "abcd";
cout << cadena << endl;
```

Este ejemplo define la matriz de caracteres *cadena* con 10 elementos (*cadena[0]* a *cadena[9]*) y asigna al primer elemento el carácter ‘a’, al segundo el carácter ‘b’, al tercero el carácter ‘c’, al cuarto el carácter ‘d’ y al quinto y siguientes el carácter nulo (valor ASCII 0 o secuencia de escape \0), con el que C++ finaliza todas las cadenas de caracteres de forma automática. Por lo tanto, el operador << sobre el objeto **cout** dará como resultado la cadena *abcd*. Otro ejemplo:

```
char cadena[] = "abcd";
cout << cadena << endl;
```

Este otro ejemplo es igual que el anterior, excepto que ahora la matriz *cadena* tiene justamente cinco elementos dispuestos como indica la figura siguiente:

	a	b	c	d	\0												
--	---	---	---	---	----	--	--	--	--	--	--	--	--	--	--	--	--

Leer y escribir una cadena de caracteres

En el capítulo *Entrada y salida estándar*, cuando se expusieron los flujos de entrada, vimos que una forma de leer una cadena de caracteres del flujo **cin** era utilizando el operador >>. Por ejemplo, si queremos leer un nombre de 40 caracteres de longitud máxima, deberemos primero definir la matriz y después leerla, así:

```
char nombre[41];
cin >> nombre;
cout << nombre << endl;
```

Ahora bien, si ejecuta las sentencias anteriores y realiza una entrada como la siguiente,

Francisco Javier

no se sorprenda cuando al visualizar la cadena vea que sólo se escribe *Francisco*. Recuerde que el operador `>>` sobre **cin** lee datos delimitados por espacios en blanco. Hay una solución más sencilla a este problema que es utilizar el método **getline** de la clase **basic_istream** cuya sintaxis es la siguiente:

```
basic_istream& getline(char_type *s, streamsize c, char_type d = '\n');
```

El método **getline** lee una cadena de caracteres desde un flujo de entrada y la almacena en *cadena*. Se entiende por cadena la serie de caracteres que va desde la posición actual de lectura en el búfer asociado con el flujo, hasta el final del flujo, hasta el primer carácter *d*, el cual se desecha, o bien hasta que el número de caracteres leídos sea igual a $n-1$; en este último caso el estado del flujo es puesto a **failbit**. La terminación `'\0'` es añadida automáticamente a la cadena leída y el carácter `'\n'`, si se encontró, es eliminado. Obsérvese que el parámetro tercero (carácter delimitador) es opcional y que por omisión vale `'\n'`. Por ejemplo:

```
char cadena[LONG_CAD];           // matriz de LONG_CAD caracteres
cin.getline(cadena, LONG_CAD); // leer cadena
```

La utilización de matrices de caracteres para la solución de problemas puede ser ampliamente sustituida por objetos de la clase **string**. La gran cantidad y variedad de métodos aportados por esta clase facilitarán enormemente el trabajo con cadenas de caracteres, puesto que, como ya sabemos, un objeto **string** encapsula una cadena de caracteres.

String

La clase **string**, basada en la plantilla **basic_string**, que pertenece al espacio de nombres **std**, está declarada en el fichero de cabecera `<string>` y proporciona métodos para examinar caracteres individuales de una cadena de caracteres, comparar cadenas, buscar y extraer subcadenas, copiar cadenas, concatenar cadenas, etc. A continuación veremos algunos de los métodos más comunes. También existe la clase **wstring** para cadenas de caracteres extendidos.

Constructores

En el capítulo *Clases* veremos que toda clase tiene al menos un método predeterminado especial denominado igual que ella, que es invocado automáticamente cada vez que se crea un objeto de esa clase; se trata del *constructor* de la clase. La clase **string** proporciona múltiples formas de su constructor **string**; la más útil quizás sea la aquí expuesta, que permite crear un objeto **string** a partir de una matriz de caracteres. Por ejemplo:

```
char cadena[40]; // matriz de 40 caracteres
```

```
// ...
string str(cadena);
cout << "Texto introducido: " << str;
```

Iteradores

Los iteradores se utilizan para navegar a través de los contenedores sin necesidad de conocer el tipo utilizado para identificar los elementos. Los iteradores **begin/end** ofrecen los elementos en el orden normal (0, 1, 2, ..., n-2, n-1); **begin** apunta al primer elemento de la secuencia y **end** al elemento siguiente al último. Los iteradores **rbegin/rend** ofrecen los elementos en el orden inverso (n-1, n-2, ..., 1, 0); **rbegin** apunta al primer elemento de la secuencia inversa y **rend** al elemento siguiente al último.

```
string str(cadena);
// ...
string::reverse_iterator e; // e es un iterador
for (e = str.rbegin(); e != str.rend(); e++)
    cout << *e; // mostrar la cadena en orden inverso
```

Acceso a un carácter

Un carácter del objeto **string** puede ser accedido utilizando el operador de indexación (`[]`) o el método **at**. El operador `[]` devuelve el carácter del objeto **string**, que está en la posición especificada. Como el índice del primer carácter es el 0, la posición especificada tiene que estar entre los valores 0 y **length()** - 1, de lo contrario el resultado será impredecible. Por ejemplo:

```
string str1 = "abcdefgh";
char car = str1[2]; // car = 'c'
```

El método **at** realiza la misma operación, pero en el caso de que el índice del carácter a obtener esté fuera de límites, lanzará una excepción.

```
string str1 = "abcdefgh";
char car = str1.at(2); // car = 'c'
```

Asignación

A un objeto **string** se le puede asignar otro objeto **string**, una matriz de caracteres e incluso un carácter. Esto puede hacerse utilizando el operador `=`, o bien el método **assign**. Por ejemplo:

```
string str1 = "abcdefgh";
string str2, str3, str4;
str2 = str1; // o bien: str2.assign(str1);
```

```
str3 = "xyz";           // o bien: str3.assign("xyz");
str4 = 'c';           // o bien: str4.assign(sizeof(char), 'c');
```

Conversiones a cadenas estilo C

Un objeto **string** puede copiarse en una cadena de caracteres estilo C (conjunto de caracteres finalizados con `'\0'` de tipo **char** *). Esto puede hacerse utilizando alguno de los métodos siguientes: **c_str**, **data** o **copy**. Por ejemplo:

```
string str1 = "abcdefgh";
const char* cadena1 = str1.c_str(); // añade '\0'
const char* cadena2 = str1.data(); // no añade '\0'

char cadena[80];
str1.copy(cadena, str1.length(), 0);
cadena[str1.length()]=0; // añadir el carácter '\0' de terminación
```

Los métodos **c_str** y **data** devuelven un puntero a una cadena de caracteres constante; **data**, a diferencia de **c_str**, no añade el carácter nulo de terminación. Estos dos métodos, en realidad, no hacen una copia, sino que devuelven la dirección de memoria de la cadena que encapsula el objeto **string**.

El método **copy** tiene tres parámetros: la cadena estilo C sobre la que se realizará la copia, el número de caracteres a copiar y el primer carácter que se copiará, que por omisión es el que está en la posición cero. Este método no añade el carácter nulo de terminación.

Comparaciones

Dos objetos **string** pueden ser comparados utilizando los operadores de relación `==`, `>`, `<`, `>=`, `<=` y `!=`. Se hace diferencia entre las letras mayúsculas y minúsculas.

En otras palabras, utilizando estos operadores es posible saber si una cadena está en orden alfabético antes (es menor) o después (es mayor) que otra y el proceso que sigue es el mismo que nosotros ejercitamos cuando lo hacemos mentalmente, comparar las cadenas carácter a carácter distinguiendo las mayúsculas de las minúsculas. El siguiente ejemplo compara dos cadenas y escribe "Abcdefg" porque esta cadena está antes por orden alfabético.

```
string str1 = "abcde", str2 = "Abcdefg";

if (str1 < str2) // si str1 es menor que str2,
    cout << str1 << endl;
else
    cout << str2 << endl;
```

Para no hacer distinción entre mayúsculas y minúsculas podemos hacer uso de las funciones **strupr** o **strlwr** de C, las cuales convierten una cadena a mayúsculas o a minúsculas, respectivamente. Por ejemplo, el resultado de ejecutar el siguiente programa es que *str1* y *str2* son iguales.

```
// Sin diferenciar mayúsculas de minúsculas
#include <iostream>
#include <string>
using namespace std;

string minusculas(string str);

int main( )
{
    string str1 = "La provincia de Santander es muy bonita";
    string str2 = "La provincia de SANTANDER es muy bonita";
    string strtemp;

    if( minusculas(str1) > minusculas(str2) )
        strtemp = "mayor que ";
    else if( minusculas(str1) < minusculas(str2) )
        strtemp = "menor que ";
    else
        strtemp = "igual a ";
    cout << str1 << " es " << strtemp << str2 << endl;
}

string minusculas(string str)
{
    char cadena[80];
    str.copy(cadena, str.length());
    cadena[str.length()]=0;
    strlwr(cadena);
    return string(cadena);
}
```

Inserción

A un objeto **string** se le puede añadir caracteres, bien al final o en otra posición cualquiera. El método **append** o el operador += permiten añadir caracteres al final y el método **insert** en cualquier otra posición. Por ejemplo:

```
string str1 = "abcd";
string str2 = "lmn";
string str3 = "efg";
str1 += str2; // str1 = "abcdlmn"
str1.insert(4, "ijk"); // str1 = "abcdijklmn"
str1.insert(4, str3); // str1 = "abcdefgijklmn"
str1.insert(7, sizeof(char), 'h'); // str1 = "abcdefghijklmn"
```

Concatenación

El operador `+` permiten concatenar (unir) dos objetos **string**. Por ejemplo:

```
string str1 = "abcd";
string str2 = "lmn";
string str3 = str1 + str2; // str3 = "abcdlmn"
```

Búsqueda

Dentro de las operaciones de búsqueda podemos elegir entre buscar una subcadena o encontrar un carácter; ambas operaciones pueden realizarse desde el principio hasta el final, o viceversa, utilizando un método **find**. A continuación se muestran algunos ejemplos:

```
int pos;
string str1 = "abcdefgh";
pos = str1.find("efg");           // pos = 4
pos = str1.rfind("efg");         // pos = 4
pos = str1.find_first_of("gfe"); // pos = 4
pos = str1.find_last_of("gfe");  // pos = 6
pos = str1.find_first_not_of("efg"); // pos = 0
pos = str1.find_last_not_of("efg"); // pos = 7
```

Este ejemplo muestra cómo buscar una cadena, o cómo buscar un carácter que esté o no en un conjunto de caracteres, empezando por el principio o por el final. Si uno de estos métodos no encuentra nada, devuelve **string::npos**.

Reemplazar

Después de identificar una subcadena o un carácter en una cadena, se puede modificar si se desea utilizando el método **replace**. Por ejemplo:

```
string str1 = "abc def gh";
str1.replace(str1.find("de"), 2, "vwxyz"); // str1 = abc vwxyzf gh
```

El primer parámetro de **replace** indica la posición desde la cual se va a iniciar la operación de reemplazar y el segundo indica cuántos caracteres a partir de la posición anterior se van a reemplazar con la cadena especificada por el tercer parámetro.

Subcadenas

El método **substr** permite obtener una subcadena de una cadena. Por ejemplo:

```
string str1 = "abc defgh ijk";
```

```
string str2 = str1.substr(5, 4); // str2 = "efgh"
```

El primer parámetro de **substr** indica la posición del primer carácter a obtener y el segundo, cuántos caracteres se van a obtener.

Tamaño

Para conocer el tamaño de una cadena se puede invocar a los métodos **size** o **length** y para modificar su tamaño, al método **resize**; en este último caso, los elementos conservados desde el primero hasta el nuevo tamaño permanecen inalterados.

```
int ncars, capacidad;
string str1 = "abc defgh ijk";
capacidad = str1.capacity();           // capacidad = 13
ncars = str1.length();                 // ncars = 13
str1.resize(str1.length() * 2);       // str1 = "abc defgh ijk"
str1 = "xyz";                          // str1 = "xyz"
capacidad = str1.capacity();           // capacidad = 26
ncars = str1.length();                 // ncars = 3
```

Para saber si una cadena está vacía, utilizar el método **empty**. Este método devuelve **true** si la cadena está vacía y **false** en caso contrario.

El método **capacity** retorna el número de caracteres máximo que podría almacenar la cadena sin necesidad de incrementar su tamaño.

Operaciones de E/S

Los operadores de entrada y salida para *basic_string* se encuentran en `<string>` en lugar de en `<iostream>`. El operador `>>` permite leer una palabra terminada por un espacio en blanco (o por un retorno de carro) expandiendo la cadena lo que se necesite para almacenar la palabra (el carácter de terminación no se lee). Ahora bien, si lo que queremos es leer una cadena formada por varias palabras separadas por espacios en blanco, entonces tenemos que utilizar la función **getline** con el siguiente formato:

```
basic_istream& getline(basic_istream&, basic_string&, char_type d = '\n');
```

Si la función **getline** intenta leer del flujo **basic_istream** y se encuentra con el final del mismo, pone el estado del flujo al valor **eofbit**.

Para escribir una cadena podemos utilizar el operador `<<`. El siguiente ejemplo clarifica lo expuesto:


```

string str;

cout << "Texto: ";
getline(cin, str); // entrada: aaa bbb
cout << str << endl; // salida: aaa bbb

cout << "Texto: ";
cin >> str; // entrada: aaa bbb
cout << str << endl; // salida: aaa

```

MATRICES MULTIDIMENSIONALES

Según lo estudiado a lo largo de este capítulo podemos decir que cada elemento de una matriz unidimensional es de un tipo primitivo. Entonces, ¿cómo procederíamos si necesitaríamos almacenar las temperaturas medias de cada día durante los 12 meses de un año?, o bien, ¿cómo procederíamos si necesitaríamos almacenar la lista de nombres de los alumnos de una determinada clase? Razonando un poco, llegaremos a la conclusión de que utilizar matrices unidimensionales para resolver los problemas planteados supondrá posteriormente un difícil acceso a los datos almacenados; esto es, responder a las preguntas: ¿cuál es la temperatura media del 10 de mayo?, o bien, ¿cuál es el nombre del alumno número 25 de la lista?, será mucho más sencillo si los datos los almacenamos en forma de tabla; en el caso de las temperaturas, una tabla de 12 filas (tantas como meses) por 31 columnas (tantas como los días del mes más largo); y en el caso de los nombres, una tabla de tantas filas como alumnos, y tantas columnas como el número de caracteres del nombre más largo. Por lo tanto, una solución fácil para los problemas planteados exige el uso de matrices de dos dimensiones.

Una matriz multidimensional, como su nombre indica, es una matriz de dos o más dimensiones.

Matrices numéricas multidimensionales

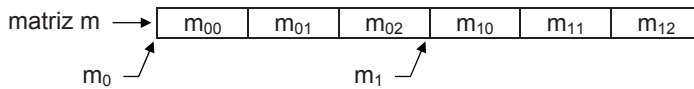
La definición de una matriz numérica de varias dimensiones se hace de la forma siguiente:

```
tipo nombre_matriz[expr-1][expr-2]....
```

donde *tipo* es un tipo primitivo entero o real. El número de elementos de una matriz multidimensional es el producto de las dimensiones indicadas por *expr-1*, *expr-2*, ... Por ejemplo, la línea de código siguiente crea una matriz de dos dimensiones con $2 \times 3 = 6$ elementos de tipo **int**:

```
int m[2][3];
```

A partir de la línea de código anterior, C++ crea una matriz bidimensional m con dos filas $m[0]$ y $m[1]$ que hacen referencia a otras dos matrices unidimensionales de tres elementos cada una. Gráficamente podemos imaginarlo así:



De la figura anterior se deduce que los elementos de una matriz bidimensional son colocados por filas consecutivas en memoria. El nombre de la matriz representa la dirección donde se localiza la primera fila de la matriz, o el primer elemento de la matriz; esto es, m , $m[0]$ y $\&m[0][0]$ son la misma dirección.

Evidentemente, el tipo de $m[0]$ y $m[1]$ es `int[]` y el tipo de los elementos de las matrices referenciadas por $m[0]$ y $m[1]$ es `int`. Además, puede comprobar la existencia y la longitud de las matrices m , $m[0]$ y $m[1]$ utilizando el código siguiente:

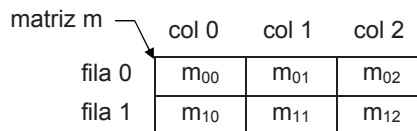
```
int m[2][3];
cout << "filas de m: " << sizeof(m)/sizeof(m[0]) << endl;
cout << "elementos de la fila 0: " << sizeof(m[0])/sizeof(int) << endl;
cout << "elementos de la fila 1: " << sizeof(m[1])/sizeof(int) << endl;
```

El resultado que se obtiene después de ejecutar el código anterior puede verlo a continuación:

```
filas de m: 2
elementos de la fila 0: 3
elementos de la fila 1: 3
```

Dicho resultado pone de manifiesto que una matriz bidimensional es en realidad una matriz unidimensional cuyos elementos son a su vez matrices unidimensionales.

Desde nuestro punto de vista, cuando se trate de matrices de dos dimensiones, es más fácil pensar en ellas como si de una tabla de f filas por c columnas se tratara. Por ejemplo:



Cuando se inicie una matriz multidimensional en el instante de su definición, el tamaño de su primera dimensión puede omitirse, ya que puede calcularse a par-

tir del resto de las dimensiones y del número total de elementos. También puede omitirse en los siguientes casos: cuando se declare como un parámetro formal en una función y cuando se haga referencia a una matriz declarada en otra parte del programa.

```
int m[][3] = {1, 2, 3, 4, 5, 6};
```

Para acceder a los elementos de la matriz m , puesto que se trata de una matriz de dos dimensiones, utilizaremos dos subíndices: el primero indicará la fila y el segundo la columna donde se localiza el elemento, según se puede observar en la figura anterior. Por ejemplo, la primera sentencia del ejemplo siguiente asigna el valor x al elemento que está en la fila 1, columna 2; y la segunda, asigna el valor de este elemento al elemento $m[0][1]$.

```
m[1][2] = x;
m[0][1] = m[1][2];
```

El cálculo que hace el compilador para saber cuántos elementos tiene que avanzar desde m para acceder a un elemento cualquiera $m[filas][cols]$ en la matriz anterior es: $fila \times elementos\ por\ fila + col$.

Una matriz numérica de varias dimensiones puede también implementarse utilizando matrices de matrices. Desde esta perspectiva, la definición de una matriz numérica de dos dimensiones puede hacerse utilizando la clase **vector** (archivo de cabecera *vector*) de la forma siguiente:

```
vector<vector<tipo> > nombre(d1, vector<tipo>(d2));
```

donde *tipo*, en el caso de matrices numéricas, es un tipo primitivo entero o real, pero en general puede ser cualquier tipo predefinido o definido por el usuario (en la expresión `vector<vector<tipo> >` hay un espacio en blanco entre `>` y `>` que desaparecerá en el próximo estándar).

El número de elementos de una matriz multidimensional en el caso de que sea rectangular es el producto de las dimensiones indicadas por $d1$ y $d2$. Por ejemplo, la línea de código siguiente crea una matriz de dos dimensiones con $2 \times 3 = 6$ elementos de tipo **int**:

```
// Matriz de 2 filas por 3 columnas representada por
// una matriz de matrices. Cada elemento de m se inicia
// con una matriz de 3 elementos de tipo int
vector<vector<int> > m(2, vector<int>(3));
// ...
m[1][2] = x;
m[0][1] = m[1][2];
```



```

do
{
    cout << "Número de filas de la matriz: ";
    cin >> filas;
}
while (filas < 1); // no permitir un valor negativo

do
{
    cout << "Número de columnas de la matriz: ";
    cin >> cols;
}
while (cols < 1); // no permitir un valor negativo

```

Después, creamos la matriz *m* con el número de filas y columnas especificado, definimos las variables *fila* y *col* para utilizarlas como subíndices fila y columna, respectivamente, y la variable *sumafila* para almacenar la suma de los elementos de una fila:

```

vector< vector<float> > m(filas);
for (int i = 0; i < filas; i++)
    m[i] = vector<float>(cols);
int fila, col;          // fila y columna del elemento accedido
float sumafila;        // suma de los elementos de una fila

```

El paso siguiente es asignar un valor desde el teclado a cada elemento de la matriz.

```

for (fila = 0; fila < filas; fila++)
    for (col = 0; col < cols; col++)
    {
        cout << "m[" << fila << "][" << col << "] = ";
        cin >> m[fila][col];
    }

```

Una vez leída la matriz, calculamos la suma de cada fila y visualizamos los resultados para comprobar el trabajo realizado.

```

for (fila = 0; fila < filas; fila++)
{
    sumafila = 0;
    for (col = 0; col < cols; col++)
        sumafila += m[fila][col];
    cout << "Suma de la fila " << fila << " = " << sumafila << endl;
}

```

El programa completo se muestra a continuación.

```
// Suma de las filas de una matriz bidimensional
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    int filas, cols; // filas y columnas de la matriz de trabajo

    do
    {
        cout << "Número de filas de la matriz:   ";
        cin >> filas;
    }
    while (filas < 1); // no permitir un valor negativo

    do
    {
        cout << "Número de columnas de la matriz: ";
        cin >> cols;
    }
    while (cols < 1); // no permitir un valor negativo

    // Matriz m de matrices de tipo float
    vector< vector<float> > m(filas);
    // Asignar a cada elemento de la matriz m
    // una matriz de "cols" elementos de tipo float
    for (int i = 0; i < filas; i++)
        m[i] = vector<float>(cols);
    float sumafila; // suma de los elementos de una fila
    int fila, col; // fila y columna del elemento accedido

    // Entrada de datos
    cout << "Introducir los valores de la matriz.\n";
    for (fila = 0; fila < filas; fila++)
        for (col = 0; col < cols; col++)
        {
            cout << "m[" << fila << "]" << col << "] = ";
            cin >> m[fila][col];
        }
    // Escribir la suma de cada fila
    for (fila = 0; fila < filas; fila++)
    {
        sumafila = 0;
        for (col = 0; col < cols; col++)
            sumafila += m[fila][col];
        cout << "Suma de la fila " << fila << " = " << sumafila << endl;
    }
    cout << "\nFin del proceso.\n";
}
```

Seguramente habrá pensado que la suma de cada fila se podía haber hecho simultáneamente a la lectura tal como se indica a continuación:

```
for (fila = 0; fila < filas; fila++)
{
    sumafila = 0;
    for (col = 0; col < cols; col++)
    {
        cout << "m[" << fila << "][" << col << "] = ";
        cin >> m[fila][col];
        sumafila += m[fila][col];
    }
    cout << "Suma de la fila " << fila << " = " << sumafila << endl;
}
```

No obstante, esta forma de proceder presenta una diferencia a la hora de visualizar los resultados, y es que la suma de cada fila se muestra a continuación de haber leído los datos de la misma.

```
Número de filas de la matriz: 2
Número de columnas de la matriz: 2
Introducir los valores de la matriz.
m[0][0] = 2
m[0][1] = 5
Suma de la fila 0 = 7
m[1][0] = 3
m[1][1] = 6
Suma de la fila 1 = 9

Fin del proceso.
```

Con este último planteamiento, una solución para escribir los resultados al final sería almacenarlos en una matriz unidimensional y mostrar posteriormente esta matriz. Este trabajo se deja como ejercicio para el lector.

Matrices de cadenas de caracteres

Las matrices de cadenas de caracteres son matrices multidimensionales, generalmente de dos dimensiones, en las que cada fila se corresponde con una cadena de caracteres. Entonces, según lo estudiado, una fila será una matriz unidimensional de tipo **char** o **unsigned char**.

Haciendo un estudio análogo al realizado para las matrices numéricas multidimensionales, la definición de una matriz de cadenas de caracteres puede hacerse de la forma siguiente:

```
[unsigned] char nombre_matriz[filas][longitud_fila];
```

Por ejemplo, la línea de código siguiente crea una matriz de cadenas de caracteres de F filas por C caracteres máximo por cada fila.

```
char m[F][C];
```

A partir de la línea de código anterior, C++ crea una matriz unidimensional m con los elementos, $m[0]$, $m[1]$, ..., $m[F-1]$, que a su vez son matrices unidimensionales de C elementos de tipo **char**. Evidentemente, el tipo de los elementos de m es **char[]** y el tipo de los elementos de las matrices referenciadas por $m[0]$, $m[1]$, ..., es **char**. Desde nuestro punto de vista, es más fácil imaginarse una matriz de cadenas de caracteres como una lista. Por ejemplo, la matriz m del ejemplo anterior estará compuesta por las cadenas de caracteres $m[0]$, $m[1]$, $m[2]$, etc.

m_0
m_1
m_2
m_3
...

Para acceder a los elementos de la matriz m , puesto que se trata de una matriz de cadenas de caracteres, utilizaremos sólo el primer subíndice, el que indica la fila. Sólo utilizaremos dos subíndices cuando sea necesario acceder a un carácter individual. Por ejemplo, la primera sentencia del ejemplo siguiente crea una matriz de cadenas de caracteres. La segunda asigna una cadena de caracteres a $m[0]$ desde el teclado; la cadena tendrá $nCarsPorFila$ caracteres como máximo y será almacenada a partir de la posición 0 de $m[0]$. Y la tercera sentencia reemplaza el último carácter de $m[0]$ por `'\0'`.

```
char m[nFilas][nCarsPorFila];
cin.getline(m[0], nCarsPorFila); // acceso a una cadena de cars.
m[0][nCarsPorFila-1] = '\0';    // acceso a un solo carácter
```

Es importante que asimile que $m[0]$, $m[1]$, etc. son cadenas de caracteres y que, por ejemplo, $m[1][3]$ es un carácter; el que está en la fila 1, columna 3.

Matrices de objetos string

C++ proporciona la clase **string** para hacer de las cadenas de caracteres objetos con sus atributos particulares, los cuales podrán ser accedidos por los métodos de dicha clase. Desde este nivel de abstracción el trabajo con matrices de cadenas de caracteres resultará mucho más sencillo.

Para ilustrar la forma de trabajar con matrices de cadenas de caracteres, vamos a realizar un programa que lea una lista de nombres y los almacene en una matriz de objetos **string**. Una vez construida la matriz, visualizaremos su contenido. La solución tendrá el aspecto siguiente:

```
Escriba los nombres que desea introducir.
Puede finalizar con eof (teclas [Ctrl][z]).
Nombre[0]: Mª del Carmen
Nombre[1]: Francisco
Nombre[2]: Javier
Nombre[3]: [Ctrl][Z]
```

```
¿Desea visualizar el contenido de la matriz? (s/n): S
```

```
Mª del Carmen
Francisco
Javier
```

La solución pasa por realizar los siguientes puntos:

- Definimos la matriz *nombre* de objetos **string**. Esto podemos hacerlo de dos formas:

```
string nombre[N]; // matriz de N elementos de tipo string
vector<string> nombre; // matriz nombre con cero elementos
```

Utilizaremos la segunda definición por la funcionalidad que nos aporta la plantilla **vector**.

- Leemos una cadena de caracteres y la añadimos a la matriz utilizando el método **push_back**. Para poder leer una cadena, utilizaremos la función **getline** declarada en `<string>`. Recuerde que esta función tiene como primer parámetro un objeto **istream** y como segundo un objeto **string**. Este proceso lo repetiremos para cada uno de los nombres que leamos.

La entrada de datos finalizará cuando se haya introducido la marca de fin de fichero. Para ello hay que tener en cuenta que si la función **getline** intenta leer del flujo **istream** y se encuentra con el final del mismo, pone el estado del flujo al valor **eofbit** lo cual puede ser detectado con su método **eof**.

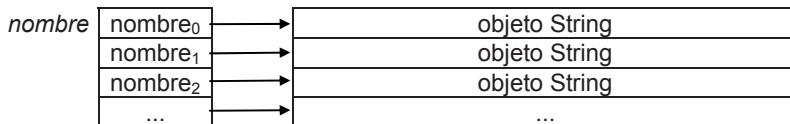
```
int fila = 0;
string nom; // un nombre
cout << "Nombre[" << fila++ << "]: ";
getline(cin, nom);
while (!cin.eof()) // si se pulsó [Ctrl][z], salir del bucle
{
    nombre.push_back(nom); // añadir una cadena
```

```

    cout << "Nombre[" << fila++ << "]: ";
    getline(cin, nom); // leer otro nombre
}

```

Gráficamente puede imaginarse el proceso descrito de acuerdo a la siguiente estructura de datos, aunque para trabajar resulte más fácil pensar en una matriz unidimensional cuyos elementos *nombre[0]*, *nombre[1]*, etc. son cadenas de caracteres.



- Una vez leídos todos los nombres deseados, los visualizamos si la respuesta a la petición de realizar este proceso es afirmativa.

El programa completo se muestra a continuación.

```

// Leer una lista de nombres
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main()
{
    // Matriz de cadenas de caracteres inicialmente vacía
    vector<string> nombre;

    cout << "Escriba los nombres que desea introducir.\n";
    cout << "Puede finalizar con eof (teclas [Ctrl][z]).\n";
    int fila = 0;
    string nom; // un nombre
    cout << "Nombre[" << fila++ << "]: ";
    getline(cin, nom);
    while (!cin.eof()) // si se pulsó [Ctrl][z], salir del bucle
    {
        nombre.push_back(nom); // añadir una cadena
        cout << "Nombre[" << fila++ << "]: ";
        getline(cin, nom); // leer otro nombre
    }
    cin.clear(); // desactivar los indicadores de error
    cout << endl;
    string respuesta;
    do
    {
        cout << "¿Desea visualizar el contenido de la matriz? (s/n): ";
        getline(cin, respuesta);
    }
}

```

```

}
while (tolower(respuesta[0]) != 's' && tolower(respuesta[0]) != 'n');
if ( tolower(respuesta[0]) == 's' )
{
    // Visualizar la lista de nombres
    cout << endl;
    for (fila = 0; fila < nombre.size(); fila++)
        cout << nombre[fila] << endl;
}
}
}

```

SENTENCIA `for_each`

La sentencia **`for_each`** es similar a la sentencia **`for`**, con la diferencia de que ahora una función se ejecuta repetidamente por cada elemento de una colección de objetos o de una matriz. Esto es, **`for_each`** es un algoritmo definido en *<algorithm>* que no hace nada sino eliminar un bucle explícito. Su sintaxis es la siguiente:

`for_each (obj.begin(), obj.end(), func);`

donde **`obj.begin`** hace referencia al primer elemento de **`obj`**, **`obj.end`** hace referencia a la posición justo después del último elemento de **`obj`** y **`func`** es el nombre de la función que se ejecutará para cada elemento de ese rango.

Por ejemplo, para mostrar el contenido de la matriz *nombre* del ejercicio anterior, podríamos escribir también el siguiente código:

```

void mostrar(string str)
{
    if (str.length() != 0) cout << str << endl;
}

int main()
{
    // ...
    if ( tolower(respuesta[0]) == 's' )
        for_each(nombre.begin(), nombre.end(), mostrar);
}

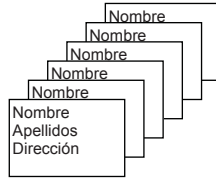
```

ESTRUCTURAS

Todas las variables que hemos utilizado hasta ahora permiten almacenar un dato y de un único tipo, excepto las matrices que almacenan varios datos pero también todos del mismo tipo. La finalidad de una estructura es agrupar una o más varia-

bles, generalmente de diferentes tipos, bajo un mismo nombre para hacer más fácil su manejo.

El ejemplo típico de una estructura es una ficha que almacena datos relativos a una persona, como *Nombre*, *Apellidos*, *Dirección*, etc. En otros compiladores, este tipo de construcciones son conocidas como *registros*.



Algunos de estos datos podrían ser a su vez estructuras. Por ejemplo, la *fecha de nacimiento* podría ser una estructura con los datos *día*, *mes* y *año*.

Definir una estructura

Para crear una estructura hay que definir un nuevo tipo de datos y declarar una variable de este tipo. La declaración de un tipo estructura incluye tanto los elementos que la componen como sus tipos. Cada elemento de una estructura recibe el nombre de *miembro* (o bien *campo* si hablamos de registros). Por ejemplo:

```
struct tficha // declaración del tipo de estructura tficha
{
    char nombre[40];
    char direccion[40];
    long telefono;
};
```

La palabra reservada **struct** indica al compilador que se está definiendo una estructura; *tficha* es un identificador que nombra el nuevo tipo definido; y cada miembro puede ser de un tipo primitivo (**char**, **int**, **float**, etc.) o un tipo derivado: matriz, puntero, unión, estructura o función.

Después de definir un tipo de estructura, podemos declarar variables de ese tipo así:

```
tficha var1, var2;
```

Este ejemplo define las variables *var1* y *var2* del tipo *tficha* definido en el ejemplo anterior; por lo tanto, *var1* y *var2* son estructuras de datos con los miembros *nombre*, *dirección* y *teléfono*.

Un miembro de una estructura se utiliza exactamente igual que cualquier otra variable. Para acceder a cualquiera de ellos se utiliza el operador punto. Por ejemplo:

```
var1.telefono = 232323;          // teléfono de var1
cin.getline(var1.nombre, 40); // nombre de var1
var2.telefono = 332343;        // teléfono de var2
cin.getline(var2.nombre, 40); // nombre de var2
```

La primera sentencia del ejemplo anterior asigna el valor 232323 al miembro *teléfono* de *var1* y la segunda lee de la entrada estándar información para el miembro *nombre* de la estructura *var1*. Las dos sentencias siguientes realizan la misma operación pero sobre *var2*.

Un miembro de una estructura puede ser a su vez otra estructura de otro tipo, o bien un puntero a la misma u otra estructura. Por ejemplo:

```
struct fecha
{
    int dia, mes, anyo;
};

struct ficha
{
    char nombre[40];
    char direccion[40];
    long telefono;
    fecha fecha_nacimiento;
};

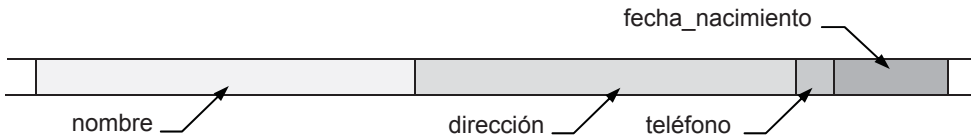
ficha persona;
```

Este ejemplo define la estructura *persona*, en la que el miembro *fecha_nacimiento* es a su vez una estructura. En este caso, si quisiéramos acceder, por ejemplo, al miembro *anyo* de *persona* tendríamos que escribir:

```
persona.fecha_nacimiento.anyo
```

Observe que *persona.fecha_nacimiento* es una estructura de tipo *fecha*; esto es, una estructura formada por los miembros *día*, *mes*, *anyo* de tipo **int**.

Finalmente, decir que los miembros de una estructura son almacenados secuencialmente byte a byte, en el mismo orden en el que son declarados. Vea la figura siguiente:



Una variable que sea una estructura permite las siguientes operaciones:

- Iniciarla en el momento de definirla:

```
ficha persona =
    { "Francisco", "Santander 1", 232323, 25, 8, 1982 };
```

- Obtener su dirección mediante el operador **&**:

```
ficha *ppersona = &persona;
```

- Acceder a uno de sus miembros:

```
long tel = persona.telefono;
```

- Asignar una estructura a otra utilizando el operador de asignación:

```
ficha otra_persona;
// ...
otra_persona = persona;
```

Cuando se asigna una estructura a otra estructura se copian uno a uno todos los miembros de la estructura fuente en la estructura destino, independientemente de cuál sea el tipo de los miembros; esto es, se duplica la estructura.

Por ejemplo, el siguiente programa define la estructura *persona* del tipo *ficha*, asigna los datos introducidos a través del teclado a cada uno de sus miembros, copia la estructura *persona* en otra estructura *otra_persona* del mismo tipo y visualiza en pantalla los datos almacenados en esta última estructura.

```
// Operaciones con estructuras
#include <iostream>
using namespace std;

struct tfecha
{
    int dia, mes, anyo;
};

struct tficha
{
    string nombre;
    string direccion;
    long telefono;
    tfecha fecha_nacimiento;
};
```

```
int main()
{
    tficha persona, otra_persona;

    // Introducir datos
    cout << "Nombre:      "; getline(cin, persona.nombre);
    cout << "Dirección:     "; getline(cin, persona.direccion);
    cout << "Teléfono:      "; cin >> persona.telefono;
    cout << "Fecha de nacimiento:\n";
    cout << "  Día:        "; cin >> persona.fecha_nacimiento.dia;
    cout << "  Mes:       "; cin >> persona.fecha_nacimiento.mes;
    cout << "  Año:      "; cin >> persona.fecha_nacimiento.anyo;

    // Copiar una estructura en otra
    otra_persona = persona;

    // Escribir los datos de la nueva estructura
    cout << "\n\n";
    cout << "Nombre:      " << otra_persona.nombre << '\n';
    cout << "Dirección:  " << otra_persona.direccion << '\n';
    cout << "Teléfono:   " << otra_persona.telefono << '\n';
    cout << "Fecha de nacimiento:\n";
    cout << "  Día:     " << otra_persona.fecha_nacimiento.dia << '\n';
    cout << "  Mes:    " << otra_persona.fecha_nacimiento.mes << '\n';
    cout << "  Año:    " << otra_persona.fecha_nacimiento.anyo << '\n';
}
```

Ejecución del programa:

```
Nombre:      Javier
Dirección:   Paseo de Pereda 10, Santander
Teléfono:    942232323
Fecha de nacimiento:
  Día:       12
  Mes:       7
  Año:       1987
```

```
Nombre:      Javier
Dirección:   Paseo de Pereda 10, Santander
Teléfono:    942232323
Fecha de nacimiento:
  Día:       12
  Mes:       7
  Año:       1987
```

Matrices de estructuras

Cuando los elementos de una matriz son de algún tipo de estructura, la matriz recibe el nombre de *matriz de estructuras* o matriz de registros. Ésta es una construcción muy útil y potente ya que nos permite manipular los datos en bloques que en muchos casos se corresponderán con objetos, en general, de la vida ordinaria.

Para definir una matriz de estructuras, primero hay que declarar un tipo de estructura que coincida con el tipo de los elementos de la matriz. Por ejemplo:

```
struct tficha
{
    string nombre;
    float nota;
};
```

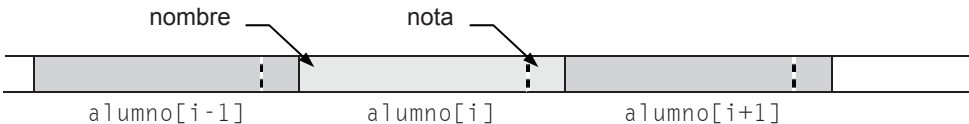
y después, se define la matriz análogamente a como se muestra a continuación:

```
vector<tficha> alumno;
```

Este ejemplo define la matriz de estructuras denominada *alumno* con 0 elementos. Para añadir los elementos *alumno[0]*, *alumno[1]*, ..., *alumno[i]*, etc., podemos utilizar el método **push_back**.

Para acceder al *nombre* y a la *nota* del elemento *i* de la matriz utilizaremos la notación:

```
alumno[i].nombre
alumno[i].nota
```



Por ejemplo, para aplicar lo expuesto hasta ahora vamos a realizar un programa que lea una lista de alumnos y las notas correspondientes a una determinada asignatura; el resultado será el tanto por ciento de los alumnos aprobados y suspendidos. Los pasos a seguir para realizar este programa pueden ser:

- Declarar el tipo de la estructura y definir la matriz de estructuras.
- Establecer un bucle para leer y almacenar en la matriz el *nombre* y la *nota* de cada alumno.

- Establecer un bucle para recorrer todos los elementos de la matriz y contar los aprobados (*nota* mayor o igual que 5) y los suspendidos (el resto).

```
for (i = 0; i < n; i++)
    if (alumno[i].nota >= 5)
        aprobados++;
    else
        suspendidos++;
```

- Escribir el tanto por ciento de aprobados y suspendidos.

El programa completo se muestra a continuación.

```
// Calcular el % de aprobados y suspendidos
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
using namespace std;

struct tficha // estructura de datos tficha
{
    string nombre;
    float nota;
};

int main()
{
    vector<tficha> alumno; // matriz de estructuras o registros
                          // inicialmente vacía

    // Entrada de datos
    cout << "Introducir datos. ";
    cout << "Para finalizar teclear la marca de fin de fichero\n\n";

    tficha un_alumno;
    cout << "Nombre: "; getline(cin, un_alumno.nombre);
    while (!cin.eof())
    {
        cout << "Nota: "; cin >> un_alumno.nota;
        cin.ignore(); // eliminar el carácter \n
        alumno.push_back(un_alumno); // añadir un alumno a la matriz
        // Siguiendo alumno
        cout << "Nombre: "; getline(cin, un_alumno.nombre);
    }

    // Contar los aprobados y suspendidos
    int aprobados = 0, suspendidos = 0;
    size_t n = 0, i = 0;
    n = alumno.size();
```

```
for (i = 0; i < n; i++)
    if (alumno[i].nota >= 5)
        aprobados++;
    else
        suspendidos++;

// Escribir resultados
cout << setprecision(4);
cout << "Aprobados:  "
    << static_cast<float>(aprobados)/n*100 << "  %\n";
cout << "Suspendidos:  "
    << static_cast<float>(suspendidos)/n*100 << "  %\n";
}
```

Ejecución del programa:

Introducir datos. Para finalizar teclear la marca de fin de fichero

```
Nombre: Elena
Nota: 10
Nombre: Pedro
Nota: 4
Nombre: Patricia
Nota: 7
Nombre: Daniel
Nota: 5
Nombre: Irene
Nota: 3
Nombre: Manuel
Nota: 6
Nombre: ^Z
Aprobados: 66.67 %
Suspendidos: 33.33 %
```

Como las variables *aprobados* y *suspendidos* son enteras, para hacer los cálculos del tanto por ciento de aprobados y suspendidos tendremos que convertir explícitamente estas variables al tipo **float** con el fin de que los cálculos se hagan en esta precisión. Si no se hace esa conversión explícita, el cociente de la división de enteros que interviene en los cálculos dará siempre 0, excepto cuando el número de aprobados sea *n*, que dará 1, o el número de suspendidos sea *n*, que también dará 1.

UNIONES

Una *unión* es una región de almacenamiento compartida por dos o más miembros generalmente de diferentes tipos. Esto permite manipular diferentes tipos de datos utilizando una misma zona de memoria, la reservada para la variable *unión*.

La declaración de una unión tiene la misma forma que la declaración de una estructura, excepto que en lugar de la palabra reservada **struct** se utiliza la palabra reservada **union**. Por lo tanto, todo lo expuesto para las estructuras es aplicable a las uniones, con la excepción de que los miembros de una *unión* no tienen cada uno su propio espacio de almacenamiento, sino que todos comparten un único espacio de tamaño igual al del miembro de mayor longitud en bytes. Por ejemplo:

```
union tmes
{
    char cmes[12];
    int nmes;
    float temperatura;
};

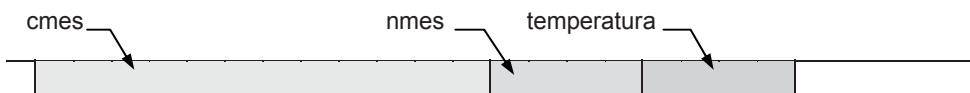
union tmes var1;
```

Este declara una variable *var1* de tipo **union tmes** que puede almacenar una cadena de caracteres, un entero o un real, pero sólo un dato, no los tres a la vez; o dicho de otra forma, *var1* almacena un dato que puede ser procesado como una cadena de caracteres, como un entero o como un real, según el miembro que se utilice para acceder al mismo.

En la figura siguiente se observa que los miembros correspondientes a *var1* no tienen cada uno su propio espacio de almacenamiento, sino que todos comparten un único espacio de tamaño igual al del miembro de mayor longitud en bytes que en este caso es *cmes* (12 bytes). Este espacio permitirá almacenar una cadena de 12 caracteres incluido el '\0', un entero o un real en simple precisión.



Si *var1* fuera una estructura en lugar de una unión, se requeriría, según se observa en la figura siguiente, un espacio de memoria igual a 12+4+4 bytes, suponiendo que un **int** ocupa 4 bytes.



Resumiendo, en una unión todos los miembros comparten el mismo espacio de memoria. El valor almacenado es sobrescrito cada vez que se asigna un valor al mismo miembro o a un miembro diferente.

Según lo expuesto, ¿qué explicación tiene el resultado de este otro programa?

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

union tunion
{
    float a;
    int b;
};

int main( )
{
    tunion var1;

    var1.a = 10.5;
    cout << var1.a << endl;
    cout << var1.b << endl;
}
```

Ejecución del programa:

```
10.5
1093140480
```

En este ejemplo la unión tiene también dos miembros: uno de tipo **float** y otro de tipo **int**; por lo tanto, el espacio de memoria compartido es de cuatro bytes. En ese espacio de memoria, después de ejecutar la sentencia `var1.a = 10.5` hay almacenado el siguiente valor expresado en binario:

```
01000001001010000000000000000000
```

Como los dos miembros son de cuatro bytes, este valor accedido desde `var1.a` de tipo **float** y visualizado dará como resultado `10.5`, pero accedido desde `var1.b` de tipo **int** y visualizado dará como resultado `1093140480`, resultados que usted mismo podrá verificar si realiza las conversiones manualmente a los formatos respectivos de **float** e **int**.

EJERCICIOS RESUELTOS

1. Realizar un programa que lea y almacene una lista de valores introducida por el teclado. Una vez leída, buscará los valores máximo y mínimo, y los imprimirá.

La solución de este problema puede ser de la siguiente forma:

- Definimos la matriz que va a contener la lista de valores y el resto de las variables necesarias en el programa.

```
vector<float> dato; // matriz de datos vacía
```

- A continuación leemos los valores que forman la lista.

```
int i = 0;
cout << "dato[" << i++ << "] = ";
float valor = leerDato();
while (!cin.eof())
{
    dato.push_back(valor);
    cout << "dato[" << i++ << "] = ";
    valor = leerDato();
}
```

- Una vez leída la lista de valores, calculamos el máximo y el mínimo. Para ello suponemos inicialmente que el primer valor es el máximo y el mínimo (como si todos los valores fueran iguales). Después comparamos cada uno de estos valores con los restantes de la lista. El valor de la lista comparado pasará a ser el nuevo mayor si es más grande que el mayor actual y pasará a ser el nuevo menor si es más pequeño que el menor actual.

```
max = min = dato[0];

for (i = 0; i < nElementos; i++)
{
    if (dato[i] > max)
        max = dato[i];
    if (dato[i] < min)
        min = dato[i];
}
```

- Finalmente, escribimos el resultado.

```
cout << "Valor máximo: " << max
      << ", valor mínimo: " << min << endl;
```

El programa completo se muestra a continuación.

```
// Valor máximo y mínimo de una lista
#include <iostream>
#include <limits>
#include <vector>
using namespace std;

float leerDato();
```

```
int main()
{
    vector<float> dato;    // matriz de datos vacía

    // Entrada de datos
    cout << "Introducir datos. Finalizar con eof (Ctrl+z).\n";
    int i = 0;
    cout << "dato[" << i++ << "] = ";
    float valor = leerDato();
    while (!cin.eof())
    {
        dato.push_back(valor);
        cout << "dato[" << i++ << "] = ";
        valor = leerDato();
    }

    // Encontrar los valores máximo y mínimo
    int nElementos = dato.size(); // número de elementos de la matriz
    float max, min;               // valor máximo y valor mínimo
    if (!dato.empty())
    {
        max = min = dato[0];
        for (i = 0; i < nElementos; i++)
        {
            if (dato[i] > max)
                max = dato[i];
            if (dato[i] < min)
                min = dato[i];
        }
        // Escribir resultados
        cout << "Valor máximo: " << max
              << ", valor mínimo: " << min << endl;
    }
    else
        cout << "No hay datos.\n";
}

float leerDato()
{
    float dato = 0;
    cin >> dato;
    if (cin.eof()) return -1; // fin de la entrada
    while (cin.fail()) // si el dato es incorrecto, limpiar el
    {
        // búfer y volverlo a leer
        cout << '\a';    // bip
        cin.clear();    // desactivar los indicadores de error
        cin.ignore(numeric_limits<int>::max(), '\n');
        cin >> dato;
    }
}
```

```

// Eliminar posibles caracteres sobrantes
cin.ignore(numeric_limits<int>::max(), '\n');
return dato;
}

```

Ejecución del programa:

Introducir datos. Finalizar con eof (Ctrl+z).

```

dato[0] = 87
dato[1] = 45
dato[2] = 68
dato[3] = 1
dato[4] = 23
dato[5] = 90
dato[6] = 7
dato[7] = 52
dato[8] = ^Z
Valor máximo: 90, valor mínimo: 1

```

- Una aplicación de las uniones puede ser definir estructuras de datos con un conjunto de miembros variable. Esto es, una estructura que permita utilizar unos miembros u otros en función de las necesidades del programa. Para ello, alguno de sus miembros tiene que ser una unión.

Por ejemplo, supongamos que deseamos diseñar una ficha para almacenar datos relativos a los libros o revistas científicas de una biblioteca. Por cada libro o revista, figurará la siguiente información:

- Número de referencia.
- Título.
- Nombre del autor.
- Editorial.
- Clase de publicación (libro o revista).
- Número de edición (sólo libros).
- Año de publicación (sólo libros).
- Nombre de la revista (sólo revistas).

Está claro que cada ficha contendrá siempre los miembros 1, 2, 3, 4 y 5 y además, si se trata de un libro, los miembros 6 y 7, o si se trata de una revista, el miembro 8. Esta disyunción da lugar a una unión con dos miembros: una estructura con los miembros 6 y 7 y el miembro 8. Veamos:

```

struct tficha // estructura variable
{
    unsigned int numref;
    string titulo;
    string autor;
}

```

```
string editorial;
enum clase libro_revista;
union
{
    struct
    {
        unsigned int edicion;
        unsigned int anyo;
    } libros;
    char nomrev[30]; // un miembro de una union no puede ser
} lr;                // un objeto string
};
```

Como aplicación de lo expuesto vamos a realizar un programa que utilizando la estructura *tficha* anterior permita:

- Almacenar en una matriz la información correspondiente a la biblioteca.
- Listar dicha información.

La estructura del programa constará de las funciones siguientes:

- a) Una función principal **main** que llamará a una función *leer* para introducir los datos que almacenarán los elementos de la matriz, y a una función *escribir* para visualizar todos los elementos de la misma.
- b) Una función *leer* con el prototipo siguiente:

```
void leer(vector<tficha>&);
```

Esta función recibe como parámetro la matriz donde hay que almacenar los datos de los libros o revistas leídos. Cada vez que se introduzcan los datos de un libro o revista, la función visualizará un mensaje preguntando si se quieren introducir más datos.

- c) Una función *escribir* con el prototipo siguiente:

```
void escribir(vector<tficha>&);
```

Esta función recibirá como parámetro la matriz cuyos elementos hay que visualizar. Cada vez que se visualice un libro o una revista se mostrará un mensaje que diga “Pulse <Entrar> para continuar” de forma que al pulsar la tecla *Entrar* se visualice el siguiente elemento de la matriz.

El programa completo se muestra a continuación. Observe que las funciones dependen sólo de sus parámetros.


```
// Biblioteca: libros y revistas científicas
#include <iostream>
#include <string>
#include <vector>
#include <limits>
using namespace std;

enum clase // tipo enumerado
{
    libro, revista
};

struct tficha // estructura variable
{
    unsigned int numref;
    string titulo;
    string autor;
    string editorial;
    enum clase libro_revista;
    union
    {
        struct
        {
            unsigned int edicion;
            unsigned int anyo;
        } libros;
        char nomrev[30];
    } lr;
};

// Prototipos de las funciones
void escribir(vector<tficha>&);
void leer(vector<tficha>&);
unsigned int leerDato();

int main() // función principal
{
    vector<tficha> biblioteca; // matriz de estructuras
    cout << "Introducir datos.\n";
    leer(biblioteca);
    cout << "\n\nListado de libros y revistas\n";
    escribir(biblioteca); // listar todos los libros y revistas
}

void leer(vector<tficha>& bibli)
{
    // Función para leer los datos de los libros y revistas
    unsigned int clase;
    char resp = 's';
```

```
tficha ficha;

while( tolower(resp) == 's')
{
    cout << "\nNúmero de refer. "; ficha.numref = leerDato();
    cout << "Título          "; getline(cin, ficha.titulo);
    cout << "Autor            "; getline(cin, ficha.autor);
    cout << "Editorial         "; getline(cin, ficha.editorial);
    do
    {
        cout << "Libro o revista (0 = libro, 1 = revista) ";
        clase = leerDato();
    }
    while (clase != 0 && clase != 1);
    if (clase == libro)
    {
        ficha.libro_revista = libro;
        cout << "Edición          ";
        ficha.lr.libros.edicion = leerDato();
        cout << "Año de public.   ";
        ficha.lr.libros.anyo = leerDato();
    }
    else
    {
        ficha.libro_revista = revista;
        cout << "Nombre revista   "; cin.getline(ficha.lr.nomrev, 30);
    }
    bibli.push_back(ficha);

    do
    {
        cout << "\n¿Más datos a introducir? s/n ";
        resp = cin.get(); cin.ignore();
    }
    while( tolower(resp) != 's' && tolower(resp) != 'n' );
}

}

void escribir(vector<tficha>& bibli)
{
    // Función para listar todos los elementos de la matriz
    int n = bibli.size(); // número de elementos de la matriz
    for (int k = 0; k < n; k++)
    {
        cout << '\n' << bibli[k].numref << ' ' << bibli[k].titulo << '\n';
        cout << bibli[k].autor << " - Ed. " << bibli[k].editorial << '\n';

        switch (bibli[k].libro_revista)
        {
            case libro :

```

```

        cout << "Edición " << bibli[k].lr.libros.edicion
            << " - año " << bibli[k].lr.libros.anyo << '\n';
        break;
    case revista :
        cout << bibli[k].lr.nomrev << '\n';
    }
    cout << "\nPulse <Entrar> para continuar ";
    cin.get();
}
}

unsigned int leerDato()
{
    unsigned int dato = 0;
    cin >> dato;
    while (cin.fail()) // si el dato es incorrecto, limpiar el
    {                 // búfer y volverlo a leer
        cout << '\a'; // bip
        cin.clear(); // desactivar los indicadores de error
        cin.ignore(numeric_limits<int>::max(), '\n');
        cin >> dato;
    }
    // Eliminar posibles caracteres sobrantes
    cin.ignore(numeric_limits<int>::max(), '\n');
    return dato;
}

```

Ejecución del programa:

```

Número de refer. 1001
Título           Enciclopedia de C++
Autor            Ceballos
Editorial        RA-MA
Libro o revista (0 = libro, 1 = revista) 0
Edición          1
Año de public.   2003

```

¿Más datos a introducir? s/n s

// ...

```

Listado de libros y revistas
1001 Enciclopedia de C++
Ceballos - Ed. RA-MA
Edición 1 - año 2003

```

Pulse <Entrar> para continuar

// ...

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

1) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

int main()
{
    using namespace std;
    int a[5] = {1, 2, 3, 4, 5}, i = 0;
    for (i = 0; i <= 5; i++)
        cout << a[i] << ' ';
}
```

- a) 1 2 3 4 5.
- b) Impredecible porque se accede a un elemento fuera de los límites de la matriz.
- c) No se puede ejecutar porque hay errores durante la compilación.
- d) 2 3 4 5 6.

2) ¿Qué hace el siguiente programa?

```
#include <iostream>

int main()
{
    using namespace std;
    int n = 0, i = 0;
    cin >> n;
    int a[n];

    for (i = 0; i < n; i++)
        cin >> a[i];
}
```

- a) Lee una matriz de n elementos de tipo entero.
- b) Lee una matriz de cero elementos de tipo entero.
- c) Produce un error durante la ejecución.
- d) Produce un error durante la compilación.

3) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
#include <vector>

int main()
{
```

```
using namespace std;
const int n = 5;
vector<int> a(n);

for (int i = 0; i < n; i++)
    cout << a[i] << ' ';
}
```

- Imprime basura (valores no predecibles).
 - Imprime 0 0 0 0 0.
 - Produce un error durante la compilación.
 - Ninguno de los anteriores.
- 4) ¿Cuál es el resultado del siguiente programa si tecleamos *hola* y a continuación pulsamos la tecla *Entrar*?

```
#include <iostream>

int main()
{
    using namespace std;
    char a[10], car, i = 0;

    while (i < 10 && (car = cin.get()) != '\n')
        a[i++] = car;
    cout << a << endl;
}
```

- hola seguido de 6 espacios en blanco.
 - hola.
 - Impredecible porque la cadena no finaliza con 0.
 - El programa produce un error durante la compilación.
- 5) ¿Cuál es el comportamiento del siguiente programa?

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

int main()
{
    using namespace std;
    int n;
    string a;
    cout << "Entero: "; cin >> n;
    cout << "Literal: "; getline(cin, a);
    cout << n << ' ' << a << '\n';
}
```

- a) Permite al usuario introducir un dato de tipo entero y a continuación un literal. Después los escribe.
 - b) Produce un error porque **getline** devuelve un valor que no se almacena.
 - c) Sólo permite al usuario introducir un literal. Después lo escribe.
 - d) Sólo permite al usuario introducir un entero. Después lo escribe.
- 6) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
typedef float matriz1df[10];

int main()
{
    using namespace std;
    matriz1df a = {1.0F, 2.0F};
    cout << sizeof(a) << endl;
}
```

- a) 40.
 - b) 8.
 - c) 4.
 - d) Produce un error durante la compilación porque *matriz1df* no es un tipo.
- 7) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

int main()
{
    using namespace std;
    const int n = 2;
    int a[][n] = {1, 2, 3, 4, 5, 6};
    cout << sizeof(a)/sizeof(a[0]) << endl;
}
```

- a) 1.
 - b) 2.
 - c) 3.
 - d) Produce un error durante la compilación.
- 8) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

struct tdato
{
    int a = 0;
    float b = 0;
};
```

```
int main()
{
    using namespace std;
    tdato s;
    cout << s.a << ' ' << s.b << endl;
}
```

- a) 0 0.
- b) Imprime basura.
- c) Produce un error durante la compilación porque la variable *s* no está bien declarada.
- d) Produce un error durante la compilación porque se han iniciado los miembros *a* y *b* a 0.

9) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
struct tdato
{
    int a;
    float b;
};

int main()
{
    using namespace std;
    tdato s = {0, 0};
    cout << s.a << ' ' << s.b << endl;
}
```

- a) 0 0.
- b) Imprime basura (valores indeterminados).
- c) Produce un error durante la compilación porque falta el nombre de la estructura.
- d) Produce un error durante la compilación porque *tdato* es una variable.

10) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>

union tdato
{
    int a;
    float b;
};

int main()
{
    using namespace std;
```

```
    tdata s;  
    s.a = 0;  
    cout << s.a << ' ' << s.b << endl;  
}
```

- a) Imprime 0 y un valor indeterminado.
 - b) 0 0.
 - c) Imprime basura (valores indeterminados).
 - d) Produce un error durante la compilación porque falta el nombre de la unión.
2. La mediana de una lista de n números se define como el valor que es menor o igual que los valores correspondientes a la mitad de los números, y mayor o igual que los valores correspondientes a la otra mitad. Por ejemplo, la mediana de:

16 12 99 95 18 87 10

es 18, porque este valor es menor que 99, 95 y 87 (mitad de los números) y mayor que 16, 12 y 10 (otra mitad).

Realizar un programa que lea un número impar de valores y dé como resultado la mediana. La entrada de valores finalizará cuando se detecte la marca de fin de fichero.

3. Analice el programa que se muestra a continuación e indique el significado que tiene el resultado que se obtiene.

```
#include <iostream>  
using namespace std;  
void visualizar( unsigned char c );  
unsigned char fnxxx( unsigned char c );  
  
int main()  
{  
    unsigned char c;  
    cout << "Introducir un carácter: ";  
    c = cin.get();  
    visualizar(c);  
    cout << "\nCarácter resultante:\n";  
    c = fnxxx(c);  
    visualizar(c);  
}  
  
void visualizar( unsigned char c )  
{  
    for (int i = 7; i >= 0; i--)  
        cout << ((c & (1 << i)) ? 1 : 0);
```



```

    cout << '\n';
}

unsigned char fnxxx( unsigned char c )
{
    return (((c)&0x01) << 7) | (((c)&0x02) << 5) |
           (((c)&0x04) << 3) | (((c)&0x08) << 1) |
           (((c)&0x10) >> 1) | (((c)&0x20) >> 3) |
           (((c)&0x40) >> 5) | (((c)&0x80) >> 7));
}

```

4. Escribir un programa que dé como resultado la frecuencia con la que aparece cada una de las parejas de letras adyacentes de un texto introducido por el teclado. No se hará diferencia entre mayúsculas y minúsculas. El resultado se presentará en forma de tabla, de la manera siguiente:

	a	b	c	d	e	f	...	z
a	0	4	0	2	1	0	...	1
b	8	0	0	0	3	1	...	0
c				.				
d				.				
e				.				
f				.				
.								
.								
.								
z								

Por ejemplo, la tabla anterior dice que la pareja de letras *ab* ha aparecido cuatro veces. La tabla resultante contempla todas las parejas posibles de letras, desde la *aa* hasta la *zz*.

Las parejas de letras adyacentes de “*hola que tal*” son: *ho*, *ol*, *la*, *a blanco* no se contabiliza por estar el carácter espacio en blanco fuera del rango ‘a’ - ‘z’, *blanco q* no se contabiliza por la misma razón, *qu*, etc.

5. Queremos escribir una función para ordenar alfabéticamente una matriz de cadenas de caracteres.

Para ello, primero diríjase al capítulo *Algoritmos*, y estudie, si aún no lo conoce, el algoritmo de ordenación basado en el “método de la burbuja”. Segundo, utilice el código del proyecto *MatrizDeString* que hemos realizado anteriormente para leer y visualizar una matriz de cadenas de caracteres. Tercero, utilizando el método de la burbuja, escriba una función *ordenar* que se ajuste al prototipo siguiente:

```
void ordenar(vector<string>& cadena);
```

El argumento *cadena* es la matriz de cadenas de caracteres que queremos ordenar alfabéticamente.

La función *ordenar* será invocada desde la función **main** una vez leída la matriz de cadenas de caracteres.

PUNTEROS, REFERENCIAS Y GESTIÓN DE LA MEMORIA

Un puntero es una variable que contiene la *dirección* de memoria de un dato o de otra variable que contiene al dato. Quiere esto decir que el puntero apunta al espacio físico donde está el dato o la variable. Un puntero puede apuntar a un objeto de cualquier tipo, como por ejemplo, a una estructura o a una función. Los punteros se pueden utilizar para referenciar y manipular estructuras de datos, para referenciar bloques de memoria asignados dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

Cuando se trabaja con punteros son frecuentes los errores por utilizarlos sin haberles asignado una dirección válida; esto es, punteros que por no estar iniciados apuntan no se sabe a dónde, produciéndose accesos a zonas de memoria no permitidas. Por lo tanto, debe ponerse la máxima atención para que esto no ocurra, iniciando adecuadamente cada uno de los punteros que utilizemos.

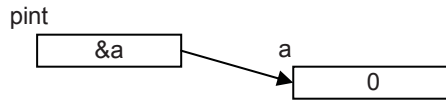
CREACIÓN DE PUNTEROS

Un puntero es una variable que guarda la dirección de memoria de otro objeto. Para declarar una variable que sea un puntero, la sintaxis es la siguiente:

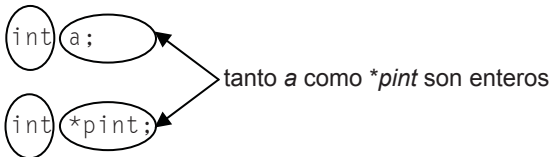
```
tipo *var-puntero;
```

En la declaración se observa que el nombre de la variable puntero, *var-puntero*, va precedido del modificador ***, el cual significa “*puntero a*”; *tipo* especifica el tipo del objeto apuntado, puede ser cualquier tipo primitivo o derivado. Por ejemplo, si una variable *pint* contiene la dirección de otra variable *a*, entonces se dice que *pint* apunta a *a*. Esto mismo expresado en código C++ es así:

```
int a = 0; // "a" es una variable entera
int *pint; // pint es un puntero a un entero
pint = &a; // pint igual a la dirección de a; entonces,
           // pint apunta al entero "a"
```



La declaración de *a* ya nos es familiar. La declaración del puntero *pint*, aunque también la hemos visto en más de una ocasión, quizás no estemos tan familiarizados con ella como con la anterior, pero si nos fijamos, ambas declaraciones tienen mucho en común. Observe la sintaxis:



Observamos que **pint* es un nemotécnico que hace referencia a un objeto de tipo *int*, por lo tanto puede aparecer en los mismos lugares donde puede aparecer un entero. Es decir, si *pint* apunta al entero *a*, entonces **pint* puede aparecer en cualquier lugar donde puede hacerlo *a*. Por ejemplo, en la siguiente tabla ambas columnas son equivalentes:

<pre>#include <iostream> using namespace std; int main() { int a = 0; a = 10; a = a - 3; cout << a << endl; }</pre>	<pre>#include <iostream> using namespace std; int main() { int a = 0, *pint = &a; *pint = 10; *pint = *pint - 3; cout << *pint << endl; }</pre>
---	---

Suponiendo definida la variable *a*, la definición:

```
int *pint = &a;
```

es equivalente a:

```
int *pint;
pint = &a;
```

En conclusión **pint* es un entero que está localizado en la dirección de memoria almacenada en *pint*.

El espacio de memoria requerido para un puntero es el número de bytes necesarios para especificar una dirección máquina, que normalmente son 4 bytes.

Un puntero iniciado correctamente siempre apunta a un objeto de un tipo particular. Un puntero no iniciado no se sabe a dónde apunta.

Operadores

Los ejemplos que hemos visto hasta ahora ponen de manifiesto que en las operaciones con punteros intervienen frecuentemente el operador *dirección de* (&) y el operador de *indirección* (*).

El operador unitario & devuelve como resultado la dirección de su operando y el operador unitario * interpreta su operando como una dirección y nos da como resultado su contenido (para más detalles, vea el capítulo *Elementos del lenguaje C++*). Por ejemplo:

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

int main()
{
    // Las dos líneas siguientes declaran la variable entera a,
    // los punteros p y q a enteros y la variable real b.
    int a = 10, *p, *q;
    double b = 0.0;
    q = &a; // asigna la dirección de a, a la variable q.
           // q apunta a la variable entera a
    b = *q; // asigna a b el valor de la variable a
    *p = 20; // error: asignación no válida
           // ¿a dónde apunta p?
    cout << "En la dirección " << q << " está el dato " << b << endl;
    cout << "En la dirección " << p << " está el dato " << *p << endl;
}
```

En teoría, lo que esperamos al ejecutar este programa es un resultado análogo al siguiente:

```
En la dirección 0x22ff6c está el dato 10
En la dirección 0x42b000 está el dato 20
```

Pero en la práctica, cuando lo ejecutamos ocurre un error por utilizar un puntero *p* sin saber a dónde apunta. Sabemos que *q* apunta a la variable *a*; dicho de otra forma, *q* contiene una dirección válida, pero no podemos decir lo mismo de *p* ya que no ha sido iniciado y, por lo tanto, su valor es desconocido para nosotros (se dice que contiene basura); posiblemente sea una dirección ocupada por el sis-

tema y entonces lo que se intentaría hacer sería sobrescribir el contenido de esa dirección con el valor 20, lo que ocasionaría graves problemas. Quizás lo vea más claro si realiza la definición de las variables así:

```
int a = 10, b = 0, *p = 0, *q = 0;
// ...
*p = 20;
```

El error se producirá igual que antes, pero ahora es claro que *p* almacena la dirección 0, dirección en la que no se puede escribir por estar reservada por el sistema.

Importancia del tipo del objeto al que se apunta

¿Cómo sabe C++ cuántos bytes tiene que asignar a una variable desde una dirección? Por ejemplo, observe el siguiente programa:

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

int main()
{
    int a = 10, *q = 0;
    double b = 0.0;
    q = &a; // q apunta al entero a
    b = *q; // asigna a b el valor de a convertido a double
    cout << "En la dirección " << q << " está el dato " << b << endl;
}
```

Se habrá dado cuenta de que en la sentencia $b = *q$, *b* es de tipo **double** y *q* apunta a un **int**. La pregunta es: ¿cómo sabe C++ cuántos bytes tiene que asignar a *b* desde la dirección *q*? La respuesta es que C++ toma como referencia el tipo del objeto definido para el puntero (**int**) y asigna el número de bytes correspondiente a ese tipo (cuatro en el ejemplo, no ocho).

OPERACIONES CON PUNTEROS

A las variables de tipo puntero, además de los operadores **&**, ***** y el operador de asignación, se les puede aplicar los operadores aritméticos **+** y **-** (sólo con enteros), los operadores unitarios **++** y **--** y los operadores de relación.

Operación de asignación

El lenguaje C++ permite que un puntero pueda ser asignado a otro puntero. Por ejemplo:

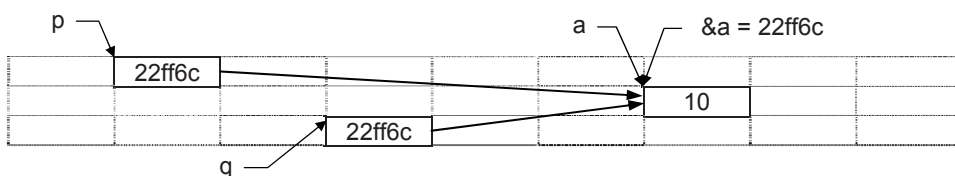
```
#include <iostream>
using namespace std;

int main()
{
    int a = 10, *p, *q;
    p = &a;
    q = p; // la dirección que contiene p se asigna a q
    cout << "En la dirección " << q << " está el valor " << *q << endl;
}
```

Ejecución del programa:

En la dirección 0x22ff6c está el valor 10

Después de ejecutarse la asignación $q = p$, p y q apuntan a la misma localización de memoria, a la variable a . Por lo tanto, a , $*p$ y $*q$ son el mismo dato; es decir, 10. Gráficamente puede imaginarse esto así:



Operaciones aritméticas

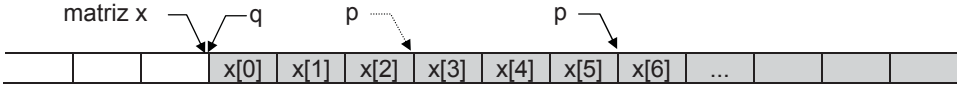
A un puntero se le puede sumar o restar un entero. La aritmética de punteros difiere de la aritmética normal en que aquí la unidad equivale a un objeto del tipo del puntero; esto es, sumar 1 implica que el puntero pasará a apuntar al siguiente objeto, del tipo del puntero, más allá del apuntado actualmente.

Por ejemplo, supongamos que p y q son variables de tipo puntero que apuntan a elementos de una misma matriz x :

```
int x[100];
int *p, *q; // declara p y q como punteros a enteros
p = &x[3]; // p apunta a x[3]
q = &x[0]; // q apunta a x[0]
```

La operación $p + n$, siendo n un entero, avanzará el puntero n enteros más allá del actualmente apuntado. Por ejemplo:

```
p = p + 3; // hace que p avance tres enteros; ahora apunta a x[6]
```



Así mismo, la operación $p - q$, después de la operación $p = p + 3$ anterior, dará como resultado 6 (elementos de tipo `int`).

La operación $p - n$, siendo n un entero, también es válida; partiendo de que p apunta a $x[6]$, el resultado de la siguiente operación será el comentado.

```
p = p - 3; // hace que p retroceda tres enteros; ahora apuntará a x[3]
```

Si p apunta a $x[3]$, $p++$ hace que p apunte a $x[4]$, y partiendo de esta situación, $p--$ hace que p apunte de nuevo a $x[3]$:

```
p++; // hace que p apunte al siguiente entero; a x[4]
p--; // hace que p apunte al entero anterior; a x[3]
```

No se permite sumar, multiplicar, dividir o rotar punteros y tampoco se permite sumarles un real.

Veamos a continuación algunos ejemplos más de operaciones con punteros. Definimos la matriz x y dos punteros pa y pb a datos de tipo entero.

```
int x[100], b, *pa, *pb;
// ...
x[50] = 10;
pa = &x[50]; // a pa se le asigna la dirección de x[50]
pb = &x[10]; // a pb se le asigna la dirección de x[10]

b = *pa + 1; // el resultado de la suma del entero *pa más 1
             // se asigna a b; es decir, b = x[50] + 1

b = *(pa + 1); // el siguiente entero al apuntado por pa,
              // es asignado a b; esto es, b = x[51]

(*pb)--; // x[10] se decrementa en una unidad

x[0] = *pb--; // a x[0] se le asigna el valor de x[10] y pb
             // pasa a apuntar al entero anterior (a x[9])
```


Comparación de punteros

Cuando se comparan dos punteros, en realidad se están comparando dos enteros, puesto que una dirección es un número entero. Esta operación tiene sentido si ambos punteros apuntan a elementos de la misma matriz. Por ejemplo:

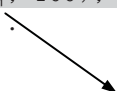
```
int n = 10, *p = 0, *q = 0, x[100];
// ...
p = &x[99];
q = &x[0];
// ...
if (q + n <= p)
    q += n;
if (q != 0 && q <= p) // 0 es una constante que identifica
    q++;              // a un puntero nulo
```

La primera sentencia **if** indica que el puntero q avanzará n elementos si se cumple que la dirección $q+n$ es menor o igual que p . La segunda sentencia **if** indica que q pasará a apuntar al siguiente elemento de la matriz si la dirección por él especificada no es nula y es menor o igual que la especificada por p .

Punteros genéricos

Un puntero a cualquier tipo de objeto puede ser convertido al tipo **void ***. Por eso, un puntero de tipo **void *** recibe el nombre de puntero genérico. Por ejemplo:

```
void fnx(void *, int);
int main()
{
    int x[100], *q = x; // se declara q y se le asigna la dirección x
                       // (el nombre de una matriz es la dirección
                       // de comienzo de la matriz)
    // ...
    fnx(q, 100); // implícitamente se convierte (int *) a (void *)
    // ...
}
void fnx(void *p, int n)
{
    // ...
}
```



En cambio, el compilador C++ no puede asumir nada sobre la memoria apuntada por un **void ***. Se necesita, por lo tanto, una conversión explícita. El operador **dynamic_cast**, como examina el tipo del objeto apuntado, no puede realizar la conversión, por lo que se tiene que utilizar **static_cast**:

```
void fnx(void *p, int n)
{
    int *q = static_cast<int *>(p);
    // ...
}
```

En este ejemplo se observa que para convertir un puntero genérico a un puntero a un **int** se ha realizado una conversión explícita (conversión *cast*).

Puntero nulo

En general, un puntero se puede iniciar como cualquier otra variable, aunque los únicos valores significativos son **0** o la dirección de un objeto previamente definido. El lenguaje C++ garantiza que un puntero que apunte a un objeto válido nunca tendrá un valor cero. El valor cero se utiliza para indicar que ha ocurrido un error; en otras palabras, que una determinada operación no se ha podido realizar.

En general, no tiene sentido asignar enteros a punteros porque quien gestiona la memoria es el sistema operativo, y por lo tanto es él el que sabe en todo momento qué direcciones están libres y cuáles están ocupadas. Por ejemplo:

```
int *px = 103825; // se inicia px con la dirección 103825
```

La asignación anterior no tiene sentido porque, ¿qué sabemos nosotros acerca de la dirección 103825?

Punteros y objetos constantes

Una declaración de un puntero precedida por **const** hace que el objeto apuntado sea una constante, no sucediendo lo mismo con el puntero. Por ejemplo:

```
char a[] = "abcd";
const char *pc = a;
pc[0] = 'z'; // error: modificación del objeto
pc = "efg"; // correcto: modificación del puntero
```

Si lo que se pretende es declarar un puntero como una constante, procederemos así:

```
char a[] = "abcd";
char* const pc = a;
pc[0] = 'z'; // correcto: modificación del objeto
pc = "efg"; // error: modificación del puntero
```

Para hacer que tanto el puntero como el objeto apuntado sean constantes, procederemos como se indica a continuación:

```
char a[] = "abcd";
const char* const pc = a;
pc[0] = 'z'; // error: modificación del objeto
pc = "efg"; // error: modificación del puntero
```

REFERENCIAS

Una referencia es un nombre alternativo (un sinónimo) para un objeto. La forma de declarar una referencia a un objeto en general es:

$$\text{tipo\& referencia} = \text{objeto}$$

Para más detalles, véase el apartado *Operador referencia a* en el capítulo *Elementos del lenguaje C++*.

No se debe aplicar la aritmética de punteros a las referencias (por ejemplo, comparar dos referencias) ni tomar la dirección de una referencia. De hecho, estas últimas operaciones no generarán un error, pero porque, como ya sabemos, dichas operaciones se realizan sobre las variables referenciadas. Por ejemplo:

```
int a[5] = {10, 20, 30, 40, 50}; // matriz a
int& rUltimo = a[4]; // referencia al último elemento de a
int& rElemento = a[0]; // referencia al primer elemento de a
int *p;

while ( rElemento <= rUltimo )
{
    p = &rElemento;
    cout << *p << ' ';
    rElemento++;
}
```

En este ejemplo, la expresión *rElemento <= rUltimo* no compara direcciones, compara los valores de *a[0]* y *a[4]*; la expresión *p = &rElemento* siempre toma la dirección de *a[0]*; la sentencia *cout << *p* siempre escribe el valor de *a[0]*; y la expresión *rElemento++* siempre incrementa el valor de *a[0]*. Según lo expuesto, el resultado al ejecutar el código anterior será *10, 11, 12, 13, ..., 50*.

Paso de parámetros por referencia

Pasar parámetros por referencia significa que lo transferido no son los valores, sino las direcciones de las variables que contienen esos valores, con lo que los

parámetros actuales se verán modificados si se modifican los contenidos de sus correspondientes parámetros formales.

Para pasar una variable por referencia, podemos utilizar una de las dos formas siguientes:

1. Pasar la dirección del parámetro actual a su correspondiente parámetro formal, el cual tiene que ser un puntero (vea *Pasando argumentos a las funciones* en el capítulo *Estructura de un programa*).

```
insertar( &ficha1 );  
// ...  
void insertar( ficha* pf )  
{  
    // La estructura de datos ficha1, apuntada por pf, puede  
    // ser modificada por esta función.  
}
```

2. Declarar el parámetro formal como una *referencia* al parámetro actual que se quiere pasar por referencia, para lo cual hay que anteponer el operador **&** al nombre del parámetro formal. Por ejemplo:

```
insertar( ficha1 );  
// ...  
void insertar( ficha& rf )  
{  
    // La estructura de datos ficha1, referenciada por rf, puede  
    // ser modificada por esta función.  
}
```

Una llamada a la función *insertar(ficha&)* como la siguiente daría lugar a un error durante la compilación, porque no se pueden definir referencias a constantes.

```
const ficha f1 = { ... };  
insertar( f1 ); // error
```

Una referencia a un objeto constante proporciona la eficiencia de los punteros en el paso de argumentos (no se hace una copia) y seguridad impidiendo que el argumento pasado se modifique, característica implícita cuando el argumento se pasa por valor. El siguiente ejemplo utiliza una referencia a una constante:

```
void insertar( const ficha& rf )  
{  
    // La estructura de datos ficha, referenciada por rf, no puede  
    // ser modificada por esta función, por ser constante.  
}
```

PUNTEROS Y MATRICES

En C++ existe una relación entre punteros y matrices tal que cualquier operación que se pueda realizar mediante la indexación de una matriz se puede hacer también con punteros.

Para clarificar lo expuesto, analicemos el siguiente programa que muestra los valores de una matriz, primero utilizando indexación y después con punteros.

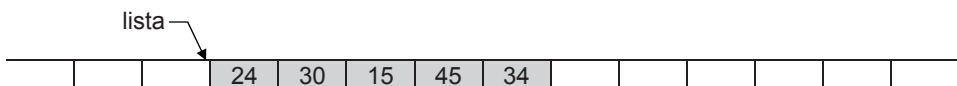
```
// Escribir los valores de una matriz.
// Versión utilizando indexación.
#include <iostream>
using namespace std;

int main()
{
    int lista[] = {24, 30, 15, 45, 34};
    for (int ind = 0; ind < 5; ind++)
        cout << lista[ind] << " ";
    cout << endl;
}
```

Ejecución del programa:

24 30 15 45 34

En este ejemplo se ha utilizado la indexación, expresión *lista[ind]*, para acceder a los elementos de la matriz *lista*. Cuando C++ interpreta esa expresión sabe que a partir de la dirección de comienzo de la matriz, esto es, a partir de *lista*, tiene que avanzar *ind* elementos para acceder al contenido del elemento especificado por ese índice. Dicho de otra forma, con la expresión *lista[ind]* se accede al contenido de la dirección *lista+ind* (ver *Operaciones aritméticas con punteros*).



Veamos ahora la versión con punteros:

```
// Escribir los valores de una matriz.
// Versión con punteros.
#include <iostream>
using namespace std;

int main()
{
    int lista[] = {24, 30, 15, 45, 34};
    int *plista = &lista[0];
```

```

for (int ind = 0; ind < 5; ind++)
    cout << *(plista+ind) << " "; // equivalente a plista[ind]
    cout << endl;
}

```

Ejecución del programa:

```
24 30 15 45 34
```

Esta versión es idéntica a la anterior, excepto que la expresión para acceder a los elementos de la matriz es ahora **(plista+ind)*.

La asignación *plista = &lista[0]* hace que *plista* apunte al primer elemento de *lista*; es decir, *plista* contiene la dirección del primer elemento, que coincide con la dirección de comienzo de la matriz; esto es, con *lista*. Por lo tanto, en lugar de la expresión **(plista+ind)*, podríamos utilizar también la expresión **(lista+ind)*. Según lo expuesto, las siguientes expresiones dan lugar a idénticos resultados:

```
lista[ind], *(lista+ind), plista[ind], *(plista+ind)
```

El mismo resultado se obtendría con esta otra versión del programa:

```

// Escribir los valores de una matriz.
// Versión con punteros.
#include <iostream>
using namespace std;

int main()
{
    int lista[] = {24, 30, 15, 45, 34};
    int *plista = lista;
    for (int ind = 0; ind < 5; ind++)
        cout << *plista++ << " ";
    cout << endl;
}

```

La asignación *plista = lista* hace que *plista* apunte al comienzo de la matriz *lista*; es decir, al elemento primero de la matriz, y la expresión **plista++* da lugar a las dos operaciones siguientes en el orden descrito: **plista*, que es el valor apuntado por *plista*, y a *plista++*, que hace que *plista* pase a apuntar al siguiente elemento de la matriz. Esto es, el bucle se podría escribir también así:

```

for (int ind = 0; ind < 5; ind++)
{
    cout << *plista << " ";
    plista++;
}

```

Sin embargo, hay una diferencia entre el identificador de una matriz y un puntero. El identificador de una matriz es una constante y un puntero es una variable. Esto quiere decir que el siguiente bucle daría lugar a un error, porque *lista* es constante y no puede cambiar de valor.

```
for (int ind = 0; ind < 5; ind++)
    cout << *lista++ << " ";
```

En cambio, un parámetro de una función que sea una matriz se considera una variable (un puntero):

```
void VisualizarMatriz(int lista[], int n)
{
    int ind;
    for (ind = 0; ind < n; ind++)
        cout << *lista++ << " ";
}
```

Otro detalle a tener en cuenta es el resultado que devuelve el operador **sizeof** aplicado a una matriz o a un puntero que apunta a una matriz:

```
int lista[] = {24, 30, 15, 45, 34};
int *plista = lista;
cout << sizeof(lista) << " " << sizeof(plista) << endl;
```

El operador **sizeof** devuelve el tamaño en bytes de su operando. Por lo tanto, aplicado a la matriz *lista* devuelve el tamaño en bytes de la matriz, en el ejemplo 20, y aplicado al puntero *plista* que apunta a una matriz devuelve el tamaño en bytes del puntero, en el ejemplo 4.

Como aplicación de lo expuesto vamos a realizar una función *CopiarMatriz* que permita copiar una matriz de cualquier tipo de datos y de cualquier número de dimensiones en otra (será una función análoga a la función **memcpy** de la biblioteca de C). Dicha función tendrá el siguiente prototipo:

```
void CopiarMatriz(void *dest, void *orig, int nbytes);
```

El parámetro *dest* es un puntero genérico que almacenará la dirección de la matriz destino de los datos y *orig* es un puntero también genérico que almacenará la dirección de la matriz origen. El tamaño en bytes de ambas matrices es proporcionado a través del parámetro *nbytes*.

Según lo estudiado anteriormente en el apartado de *Punteros genéricos*, la utilización de éstos como parámetros permitirá pasar argumentos que sean matrices de cualquier tipo. Así mismo, según lo estudiado en este apartado, el nombre de

una matriz es siempre su dirección de comienzo, independientemente del tamaño y del número de dimensiones que tenga.

¿Qué tiene que hacer la función *CopiarMatriz*? Sencillamente copiar *nbytes* desde *orig* a *dest*. Piense que una matriz, sin pensar en el tipo de datos que contiene, no es más que un bloque de bytes consecutivos en memoria. Pensando así, *orig* es la dirección de comienzo del bloque de bytes a copiar y *dest* es la dirección del bloque donde se quieren copiar. La copia se realizará byte a byte, lo que garantiza independencia del tipo de datos (el tamaño de cualquier tipo de datos es múltiplo de un byte). Esto exige, según se explicó anteriormente en este mismo capítulo en el apartado *Importancia del tipo del objeto al que se apunta*, convertir *dest* y *orig* a punteros a **char**. Según lo expuesto, la solución puede ser así:

```
void CopiarMatriz( void *dest, void *orig, int nbytes )
{
    char *destino = static_cast<char *>(dest);
    char *origen = static_cast<char *>(orig);
    for (int i = 0; i < nbytes; i++)
    {
        destino[i] = origen[i];
    }
}
```

El siguiente programa utiliza la función *CopiarMatriz* para copiar una matriz *m1* de dos dimensiones de tipo **int** en otra matriz *m2* de iguales características.

```
// Copiar una matriz en otra.
#include <iostream>
using namespace std;

void CopiarMatriz( void *dest, void *orig, int nbytes );
int main()
{
    const int FILAS = 2;
    const int COLS = 3;
    int m1[FILAS][COLS] = {24, 30, 15, 45, 34, 7};
    int m2[FILAS][COLS];

    CopiarMatriz(m2, m1, sizeof(m1));

    for (int f = 0; f < FILAS; f++)
    {
        for (int c = 0; c < COLS; c++)
            cout << m2[f][c] << " ";
        cout << endl;
    }
}
```



```

void CopiarMatriz( void *dest, void *orig, int nbytes )
{
    char *destino = static_cast<char *>(dest);
    char *origen = static_cast<char *>(orig);
    for (int i = 0; i < nbytes; i++)
    {
        destino[i] = origen[i];
    }
}

```

Ejecución del programa:

```

24 30 15
45 34 7

```

Punteros a cadenas de caracteres

Puesto que una cadena de caracteres es una matriz de caracteres, es correcto pensar que la teoría expuesta anteriormente es perfectamente aplicable a cadenas de caracteres. Un puntero a una cadena de caracteres puede definirse de alguna de las dos formas siguientes:

```

char *cadena;
unsigned char *cadena;

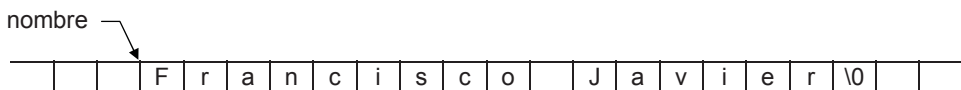
```

¿Cómo se identifica el principio y el final de una cadena? La dirección de memoria donde comienza una cadena viene dada por el nombre de la matriz que la contiene y el final, por el carácter `\0` con el que C++ finaliza todas las cadenas. El siguiente ejemplo define e inicia la cadena de caracteres *nombre*.

```

char *nombre = "Francisco Javier";
cout << nombre << endl;

```



En el ejemplo anterior *nombre* es un puntero a una cadena de caracteres. El compilador C++ asigna la dirección de comienzo del literal “Francisco Javier” al puntero *nombre* y finaliza la cadena con el carácter `\0`. Por lo tanto, el operador `<<` sabe que la cadena de caracteres que tiene que visualizar empieza en la dirección *nombre* y que a partir de aquí, tiene que ir accediendo a posiciones sucesivas de memoria hasta encontrar el carácter `\0`.

Es importante tomar nota de que *nombre* no contiene una copia de la cadena asignada, sino la dirección de memoria del lugar donde la cadena está almacena-

da, que coincide con la dirección del primer carácter; los demás caracteres, por definición de matriz, están almacenados consecutivamente. Según esto, en el ejemplo siguiente, *nombre* apunta inicialmente a la cadena de caracteres “Francisco Javier” y, a continuación, reasigna el puntero para que apunte a una nueva cadena. La cadena anterior se pierde porque el contenido de la variable *nombre* ha sido sobrescrito con una nueva dirección, la de la cadena “Carmen”.

```
char *nombre = "Francisco Javier";
cout << nombre << endl;
nombre = "Carmen";
```

Un literal, por tratarse de una constante de caracteres, no se puede modificar. Por ejemplo, si intenta ejecutar el código siguiente obtendrá un error:

```
char *nombre = "Francisco Javier";
nombre[9] = '-'; // error en ejecución
```

Lógicamente, el error comentado anteriormente no ocurre cuando la cadena de caracteres viene dada por una matriz que no haya sido declarada constante (**const**), según muestra el ejemplo siguiente:

```
char nombre[] = "Francisco Javier";
char *pnombre = nombre;
pnombre[9] = '-'; // se modifica el elemento de índice 9
```

El siguiente ejemplo presenta una función que copia una cadena en otra. Para ello, utiliza una función *copicad* que tiene dos parámetros: la matriz destino y la matriz origen que contiene la cadena a copiar (recuerde que la biblioteca de C proporciona la función **strcpy** para realizar esta misma operación). Según esto, la llamada a la función podría ser así:

```
copicad(cadena2, cadena1); // copia la cadena1 en la cadena2
```

Resulta evidente que la función *copicad* tiene que recibir como parámetros las direcciones de las matrices destino y fuente. Por lo tanto, la solución del problema planteado puede ser así:

```
// Función para copiar una cadena en otra.
#include <iostream>
using namespace std;

void copicad(char *, char *);

int main()
{
    char cadena1[81], cadena2[81];
```

```

    cout << "Introducir una cadena: ";
    cin.getline(cadena1, 81);
    copiacad(cadena2, cadena1); // copia la cadena1 en la cadena2
    cout << "La cadena copiada es: " << cadena2 << endl;
}

void copiacad(char *dest, char *orig) // copia orig en dest
{
    *dest = *orig;
    while (*orig != '\0')
    {
        dest++;
        orig++;
        *dest = *orig;
    }
}

```

Ejecución del programa:

*Introducir una cadena: hola
La cadena copiada es: hola*

Aplicando lo estudiado en operaciones con punteros, podemos escribir una nueva versión de la función *copiacad* así:

```

void copiacad(char *dest, char *orig) // copia orig en dest
{
    while (*dest++ = *orig++);
}

```

MATRICES DE PUNTEROS

En capítulos anteriores, hemos trabajado con matrices multidimensionales, aunque en la práctica lo más habitual es utilizar matrices de punteros por las ventajas que esto reporta, como verá más adelante.

Se puede definir una matriz, para que sus elementos contengan, en lugar de un dato de un tipo primitivo, una dirección o puntero. Por ejemplo:

```

int *p[5]; // matriz de 5 elementos de tipo (int *)
int b = 30; // variable de tipo int
p[0] = &b; // p[0] apunta al entero b
cout << *p[0]; // escribe 30

```

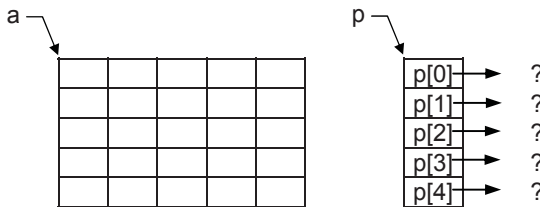
Este ejemplo define una matriz *p* de cinco elementos, cada uno de los cuales es un puntero a un **int**, y una variable entera *b*. A continuación asigna al elemento *p[0]* la dirección de *b* y escribe su contenido. Análogamente podríamos proceder

con el resto de los elementos de la matriz. Así mismo, si un elemento como $p[0]$ puede apuntar a un entero, también puede apuntar a una matriz de enteros; en este caso, el entero apuntado se corresponderá con el primer elemento de dicha matriz.

Según lo expuesto, una matriz de dos dimensiones y una matriz de punteros se pueden utilizar de forma parecida, pero no son lo mismo. Por ejemplo:

```
int a[5][5]; // matriz de dos dimensiones
int *p[5];  // matriz de punteros
```

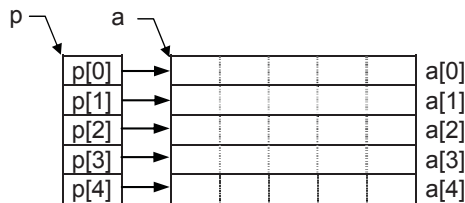
Las declaraciones anteriores dan lugar a que el compilador de C++ reserve memoria para una matriz a de 25 elementos de tipo entero y para una matriz p de cinco elementos declarados como punteros a objetos de tipo entero (**int ***).



Supongamos ahora que cada uno de los objetos apuntados por los elementos de la matriz p es a su vez una matriz de cinco elementos de tipo entero. Por ejemplo, hagamos que los objetos apuntados sean las filas de a :

```
int a[5][5]; // matriz de dos dimensiones
int *p[5];  // matriz de punteros a int

for (int i = 0; i < 5; i++)
    p[i] = a[i]; // asignar a p las filas de a
```



El bucle que asigna a los elementos de la matriz p las filas de a ($p[i] = a[i]$), ¿no podría sustituirse por una sola asignación $p = a$? No, porque los niveles de indirección son diferentes; dicho de otra forma, los tipos de p y a no son iguales ni admiten una conversión entre ellos. Veamos:

Tipo de p	<code>int * [5]</code>	(matriz de cinco elementos de tipo puntero a int)
Tipo de $p[i]$	<code>int *</code>	(puntero a int)

Tipo de a	<code>int (*)[5]</code>	(puntero a una matriz de cinco elementos de tipo <code>int</code> ; compatible con <code>int [][][5]</code> ; los elementos de a , $a[i]$, son matrices unidimensionales)
Tipo de $a[i]$	<code>int *</code>	(puntero a <code>int</code> ; compatible con <code>int []</code>)

A la vista del estudio de tipos anterior, la única asignación posible es la realizada: $p[i] = a[i]$.

El acceso a los elementos de la matriz p puede hacerse utilizando la notación de punteros o utilizando la indexación igual que lo haríamos con a . Por ejemplo, para asignar valores a los enteros referenciados por la matriz p y después visualizarlos, podríamos escribir el siguiente código:

```
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
        cin >> p[i][j];

for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
        cout << setw(7) << p[i][j];
    cout << endl;
}
```

Según lo expuesto, ¿qué diferencias hay entre p y a ? En la matriz p el acceso a un elemento se efectúa mediante una indirección a través de un puntero y en la matriz a , mediante una multiplicación y una suma. Por otra parte, como veremos más adelante, las matrices apuntadas por p pueden ser de longitud diferente, mientras que en una matriz como a todas las filas tienen que ser de la misma longitud.

Supongamos ahora que necesitamos almacenar la dirección p , dirección de comienzo de la matriz de punteros, en otra variable q . ¿Cómo definiríamos esa variable q ? La respuesta la obtendrá si responde a esta otra pregunta: ¿a quién apunta p ? Evidentemente habrá respondido: al primer elemento de la matriz de punteros; esto es, a $p[0]$ que es un puntero a un `int`. Entonces p apunta a un puntero que a su vez apunta a un entero. Por lo tanto, q tiene que ser definida para que pueda almacenar un puntero a un puntero a un `int`.

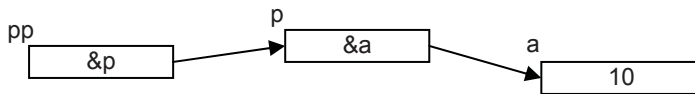
Punteros a punteros

Para especificar que una variable es un puntero a un puntero, la sintaxis utilizada es la siguiente:

```
tipo **varpp;
```

donde *tipo* especifica el tipo del objeto apuntado después de una doble indirección (puede ser cualquier tipo incluyendo tipos derivados) y *varpp* es el identificador de la variable puntero a puntero. Por ejemplo:

```
int a, *p, **pp;
a = 10; // dato
p = &a; // puntero que apunta al dato
pp = &p; // puntero que apunta al puntero que apunta al dato
```



Se dice que *p* es una variable con un nivel de indirección; esto es, a través de *p* no se accede directamente al dato, sino a la dirección que indica dónde está el dato. Haciendo un razonamiento similar diremos que *pp* es una variable con dos niveles de indirección.

El código siguiente resuelve el ejemplo anterior, pero utilizando ahora una variable *q* declarada como un puntero a un puntero. El acceso a los elementos de la matriz *a* utilizando el puntero a puntero *q* puede hacerse utilizando la indexación igual que lo haríamos con *a* o utilizando la notación de punteros. Utilizando la indexación sería así:

```
// Puntero a puntero.
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int i, j;
    int a[5][5]; // matriz de dos dimensiones
    int *p[5]; // matriz de punteros
    int **q; // puntero a puntero a un entero

    for (i = 0; i < 5; i++)
        p[i] = a[i]; // asignar a p las filas de a
    q = p;

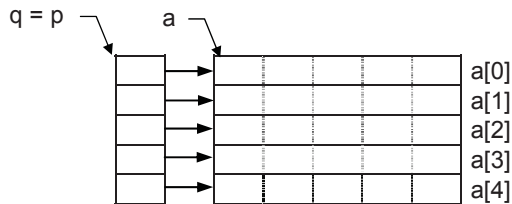
    for (i = 0; i < 5; i++)
        for (j = 0; j < 5; j++)
        {
            cout << "q[" << i << "][" << j << "]: ";
            cin >> q[i][j];
        }
}
```

```

for (i = 0; i < 5; i++)
{
    for (j = 0; j < 5; j++)
        cout << setw(7) << q[i][j];
    cout << endl;
}
}
    
```

Seguramente habrá pensado: ¿por qué no se asigna a q directamente a ($q=a$)? Pues porque q es una variable con dos niveles de indirección y a es una variable con un solo nivel de indirección. En otras palabras el tipo de q es `int **` y el tipo de a es `int (*)[5]`. En cambio, p si es una variable con dos niveles de indirección; su tipo es `int *[5]`, que significa matriz de cinco elementos de tipo `int *`; pero como el nombre de la matriz es un puntero a su primer elemento que es de tipo puntero a `int`, estamos en el caso de un puntero a un puntero.

A continuación modifiquemos el ejemplo anterior para realizar el acceso a los elementos de la matriz p utilizando la notación de punteros.



En la figura anterior podemos observar que la dirección de comienzo de la matriz es q o p . Entonces, si la dirección de comienzo de la matriz de punteros es q , suponiendo un valor entero i entre 0 y 4, ¿cuál es la dirección de su elemento i ? Evidentemente $q+i$. Y, ¿cuál es el contenido de esta dirección? Esto es, ¿cuál es el valor de $*(q+i)$ o $q[i]$? Pues la dirección de la fila $a[i]$ de la matriz a ; esto es, $q[i]$ y $a[i]$ son la misma dirección, la de la fila i de la matriz a . Si a esta dirección le sumamos un entero j entre 0 y 4 ($*(q+i)+j$ o $q[i]+j$), ¿cuál es el resultado? Pues otra dirección: la que corresponde al elemento j de la fila $a[i]$. Y, ¿cuál es el contenido de esta dirección? Esto es, ¿cuál es el valor de $*(*(q+i)+j)$ o $*(q[i]+j)$ o $q[i][j]$? Pues el valor del elemento $a[i][j]$ de la matriz a . De este análisis se deduce que las siguientes expresiones representan todas ellas el mismo valor:

$$q[i][j], *(q[i]+j), *((*(q+i)+j)$$

Según lo expuesto, observe que las direcciones $q+l$ y $*(q+l)$ tienen significados diferentes. Por ejemplo:

$q+l+2$ es la dirección del elemento $q[3]$ de la matriz de punteros.

$*(q+1)+2$ es la dirección del elemento $q[1][2]$.

$*(*(q+1)+2)$ es el valor del elemento $q[1][2]$.

De acuerdo con lo expuesto, la versión con punteros del ejemplo anterior presenta solamente la siguiente modificación:

```
for (i = 0; i < 5; i++)
  for (j = 0; j < 5; j++)
  {
    cout << "q[" << i << "][" << j << "]: ";
    cin >> (*(*(q+i)+j));
  }
```

```
for (i = 0; i < 5; i++)
{
  for (j = 0; j < 5; j++)
    cout << setw(7) << (*(*(q+i)+j));
  cout << endl;
}
```

Matriz de punteros a cadenas de caracteres

Haciendo un estudio análogo al realizado para las matrices de punteros numéricas, diremos que una matriz de punteros a cadenas de caracteres es una matriz unidimensional en la que cada elemento es de tipo **char *** o **unsigned char ***. Por ejemplo:

```
char *p[5];           // matriz de cinco elementos de tipo (char *)
char c = 'z';        // variable c de tipo char
p[0] = &c;           // p[0] apunta al carácter 'z'
cout << *p[0] << endl; // escribe: z
```

Este ejemplo define una matriz p de cinco elementos, cada uno de los cuales es un puntero a un carácter (**char ***), y una variable c de tipo **char** iniciada con el valor 'z'. A continuación asigna al elemento $p[0]$ la dirección de c y escribe su contenido. Análogamente podríamos proceder con el resto de los elementos de la matriz. Así mismo, si un elemento como $p[0]$ puede apuntar a un carácter, también puede apuntar a una cadena de caracteres o matriz unidimensional de caracteres; en este caso, el carácter apuntado se corresponderá con el primer elemento de la cadena. Por ejemplo:

```
char *p[5];           // matriz de 5 elementos de tipo (char *)
p[0] = "hola";        // p[0] apunta a la cadena hola
cout << p[0] << endl; // escribe: hola
```



```

        "agosto", "septiembre", "octubre",
        "noviembre", "diciembre"           });
    return ((mm > 0 && mm <= 12) ? mes[mm] : mes[0]);
}

int main()
{
    char *m = nombre_mes(8);
    cout << "\nMes: " << m << endl; // escribe Mes: agosto
}

```

En este ejemplo, *mes* es una matriz de 13 elementos (0 a 12) que son punteros a cadenas de caracteres. Cada elemento de la matriz ha sido iniciado con un literal. Como se ve, éstos son de diferente longitud y todos serán finalizados automáticamente por C++ con el carácter nulo. Si en lugar de utilizar una matriz de punteros, hubiéramos utilizado una matriz de dos dimensiones, el número de columnas tendría que ser el del literal más largo, más uno para el carácter nulo, con lo que la ocupación de memoria sería mayor.

Siguiendo con el ejemplo, es fácil comprobar que las declaraciones **char *mes[]** y **char **mes** no son equivalentes. La primera declara una matriz de punteros a cadenas de caracteres y la segunda un puntero a un puntero a una cadena de caracteres. Por lo tanto, sería un error sustituir en el código **mes[]* por ***mes*.

ASIGNACIÓN DINÁMICA DE MEMORIA

C++ cuenta fundamentalmente con dos métodos para almacenar información en la memoria. El primero utiliza variables globales y locales. En el caso de variables globales, el espacio es fijado para ser utilizado a lo largo de toda la ejecución del programa; y en el caso de variables locales, la asignación se hace a través de la pila del sistema; en este caso, el espacio es fijado temporalmente, mientras la variable existe. El segundo método utiliza los operadores **new** y **delete** de C++. Como es lógico, estos operadores utilizan el área de memoria libre para realizar las asignaciones de memoria solicitadas.

La *asignación dinámica de memoria* consiste en asignar la cantidad de memoria necesaria para almacenar un objeto durante la ejecución del programa, en vez de hacerlo en el momento de la compilación del mismo. Cuando se asigna memoria para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Según esto, lo que tiene que hacer el compilador es asignar una cantidad fija de memoria para almacenar la dirección del objeto asignado dinámicamente, en vez de hacer una asignación para el objeto en sí. Esto implica declarar un puntero a un tipo de datos igual al tipo del objeto que se quiere asignar dinámicamente. Por ejemplo, si queremos asignar memoria dinámicamente para

una matriz de enteros, el objeto apuntado será el primer entero, lo que implica declarar un puntero a un entero; esto es:

```
int *p;
```

Operadores para asignación dinámica de memoria

La biblioteca de C++ proporciona un operador para asignar memoria dinámicamente, **new**, y otro para liberar el espacio de memoria asignado para un objeto cuando éste ya no sea necesario, **delete**.

new

El operador **new** permite asignar un bloque de n bytes consecutivos en memoria para almacenar uno o más objetos de un tipo cualquiera. Su sintaxis es así:

```
#include <new>
void *operator new( size_t n, const std::nothrow_t& ) throw();
void *operator new[]( size_t n, const std::nothrow_t& ) throw();
void *operator new( size_t n ) throw (bad_alloc);
void *operator new[]( size_t n ) throw (bad_alloc);
```

Cuando el operador **new** se utiliza para asignar espacio para un objeto, invoca a *operator new*, y cuando se utiliza para asignar espacio para una matriz, invoca a *operator new[]*. Por ejemplo:

```
// Espacio para un objeto double
double *pc = new double;
// Espacio para una matriz de n objetos double
double *mc = new double[n];
```

Este operador devuelve un puntero que referencia el espacio asignado y si el argumento es 0, asigna un bloque de tamaño 0 bytes devolviendo también un puntero válido.

Después de invocar al operador **new** hay que verificar si ha sido posible realizar la asignación de memoria solicitada. Si hay insuficiente espacio de memoria, **new** con una especificación de excepciones vacía (**throw()** => **nothrow**) retorna un puntero nulo (valor cero). En un caso como éste, lo más probable es que no tenga sentido continuar con la ejecución del programa. Por ejemplo:

```
int *p = 0;
p = new (nothrow) int[n];
if (p == 0)
{
```

```

    cout << "Insuficiente espacio de memoria\n";
    return -1;
}

```

En cambio, si **new** con la especificación de la excepción **bad_alloc** (véase el capítulo de *Excepciones*) no encuentra memoria suficiente para asignar, lanza ésta. Por lo tanto, deberemos atraparla y tratarla. Por ejemplo:

```

int *p = 0;
try
{
    p = new int[n];
}
catch(bad_alloc e)
{
    cout << "Insuficiente espacio de memoria\n";
    exit(-1);
}

```

Observe que el tamaño en bytes del bloque que se asignará dinámicamente es $\text{sizeof(int)} * n$ y que a dicho bloque se accederá a través de p .



En este último ejemplo, como otra alternativa, se ha utilizado la función **exit** en lugar de la sentencia **return**. La función **exit** finaliza el programa; en cambio la sentencia **return** devuelve el control a la función que invocó a ésta que se está ejecutando; si la sentencia **return** pertenece a la función **main**, lógicamente el programa finaliza.

Según lo expuesto podríamos escribir una función *asignar* que permitiera obtener un espacio de memoria de n bytes, así:

```

void *asignar(int nbytes)
{
    // Asignar un bloque de memoria iniciado a cero
    char *p;
    try
    {
        p = new char[nbytes]; // asignar un bloque de nbytes
        fill(p, p+nbytes, 0); // iniciar este bloque a cero
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
    }
}

```

```

    return 0;
}
return p;
}

```

La función **fill** (alternativa a **memset** de C) declarada en *algorithm* permite iniciar un bloque de memoria (una matriz de elementos de tipo *T*). El primer argumento es la dirección del primer elemento que se desea iniciar, el segundo es la dirección del elemento siguiente al último que se desea iniciar, y el tercero es el valor de tipo *T* empleado para iniciar cada elemento.

La función *asignar* recibe como parámetro el número de bytes a asignar y devuelve un puntero al espacio de memoria asignado, o bien un cero (puntero nulo) si la asignación no se puede realizar. Una llamada a esta función podría realizarse de la forma siguiente:

```
int *p = static_cast<int *>(asignar(nbytes));
```

Recuerde que en C++ existe una conversión implícita desde un puntero a cualquier tipo al tipo **void ***, pero la conversión inversa hay que realizarla explícitamente, simplemente por seguridad, ya que en este caso el compilador no conoce el tipo del objeto al que está apuntando realmente.

delete

El operador **delete** permite liberar un bloque de memoria asignado por el operador **new**, pero no pone el puntero a **0**. Si el puntero que referencia el bloque de memoria que deseamos liberar es nulo, la función **delete** no hace nada. Su sintaxis es así:

```

#include <new>
void operator delete(void *);
void operator delete[](void *);

```

Si la memoria liberada por **delete** no ha sido previamente asignada por **new**, se pueden producir errores durante la ejecución del programa. Por ejemplo, si a un puntero le asignamos la dirección de una matriz automática (una matriz primitiva cuyo espacio de memoria se reservó durante la compilación), ese espacio de memoria no hay que liberarlo.

El siguiente ejemplo, además de asignar memoria para un objeto **double** y para una matriz de *n* objetos **double**, utiliza la función **delete** para liberar primero la memoria asignada al objeto **double** y después a la matriz; en este último caso obsérvese que debe utilizar los `[]` para indicar que se trata de una matriz; si no lo hace, el bloque de memoria no se liberará. Es un buen estilo de programación li-

berar la memoria asignada cuando ya no se necesite. En el sistema operativo Windows la memoria no liberada crea lagunas de memoria o fugas de memoria; esto es, los bloques de memoria no liberados no estarán disponibles hasta que no se reinicie la máquina.

```
// Espacio para un objeto double
double *pc = new double;
// Espacio para una matriz de n objetos double
double *mc = new double[n];
// ...
// Liberar el espacio de memoria asignado
delete pc;
delete [] mc;
```

Reasignar un bloque de memoria

En alguna ocasión necesitaremos cambiar el tamaño de un bloque de memoria previamente asignado. Para realizar esto, la biblioteca de C++ no proporciona un operador, o bien una función que realice este trabajo. No obstante, escribir una función que haga esto es una tarea fácil. Por ejemplo, la función *reasignar* que se expone a continuación, reasigna el bloque de memoria apuntado por *p* de tamaño *tactu* bytes con un nuevo tamaño *tnuevo* bytes, manteniendo los datos del espacio conservado, y devuelve un puntero al nuevo espacio.

```
void *reasignar(void *p, int tactu, int tnuevo)
{
    // Asignar un bloque de memoria iniciado a cero y
    // conservando el contenido actual.
    char *n, *v = static_cast<char *>(p);
    int tam = min(tactu, tnuevo);
    try
    {
        n = new char[tnuevo]; // asignar un nuevo bloque de memoria
        fill(n, n+tnuevo, 0); // ponerlo a cero
        if (v) copy(v, v+tam, n); // copiar tam bytes desde v a n
        delete [] v; // liberar el bloque de memoria viejo
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        return 0;
    }
    return n;
}
```

El parámetro *p* es un puntero que apunta al comienzo del bloque de memoria actual; esto es, al bloque que se quiere reasignar. Si *p* es cero, esta función se

comporta igual que **new** y asigna un nuevo bloque de *tnuevo* bytes. Si *p* no es un puntero nulo, entonces tiene que ser un puntero devuelto por el operador **new**. El argumento *tactu* da el tamaño actual en bytes del bloque que se desea reasignar. El contenido del bloque no cambia en el espacio conservado.

La función **copy** (alternativa a **memcpy** de C) declarada en *algorithm* copia un bloque de memoria en otro (elementos de una matriz de tipo *T* en otra matriz del mismo tipo). El primer argumento es la dirección del primer elemento que se desea copiar, el segundo es la dirección del elemento siguiente al último que se desea copiar, y el tercero es la dirección del primer elemento destino de los datos.

El siguiente programa, utilizando las funciones *asignar* y *reasignar* expuestas anteriormente, muestra cómo realizar una reasignación de memoria y pone de manifiesto que después de realizarla, la información no varía en el espacio de memoria conservado. Por último, el bloque de memoria es liberado.

```
// Reasignar el tamaño de una matriz
#include <iostream>
using namespace std;

void *asignar(int nbytes);
void *reasignar(void *p, int tactu, int tnuevo);

int main()
{
    int *p = 0;
    int i, n = 10;

    // Asignar memoria para una matriz de n elementos de tipo int
    if (p = static_cast<int *>(asignar(n*sizeof(int))))
        cout << "Se han asignado " << n*sizeof(int)
            << " bytes de memoria\n";
    // Asignar valores a los elementos de la matriz
    for (i = 0; i < n; i++)
        p[i] = i;
    // Mostrar los valores de los elementos de la matriz
    for (i = 0; i < n; i++)
        cout << p[i] << ' ';
    cout << endl;
    // Reasignar el tamaño de la matriz para que tenga n+5 elementos
    if (p = static_cast<int *>(
        reasignar(p, n*sizeof(int), (n+5)*sizeof(int))))
    {
        n += 5;
        cout << "Se han asignado " << n*sizeof(int)
            << " bytes de memoria\n";
    }
    // Mostrar los valores de los elementos de la matriz
```

```
    for (i = 0; i < n; i++)
        cout << p[i] << ' ';
    cout << endl;
    // Liberar la memoria asignada a la matriz
    delete [] p;
}

void *asignar(int nbytes)
{
    // Asignar un bloque de memoria iniciado a cero
    // ...
}

void *reasignar(void *p, int tactu, int t nuevo)
{
    // Asignar un bloque de memoria iniciado a cero y
    // conservando el contenido actual.
    // ...
}
```

Ejecución del programa:

```
Se han asignado 40 bytes de memoria
0 1 2 3 4 5 6 7 8 9
Se han asignado 60 bytes de memoria
0 1 2 3 4 5 6 7 8 9 0 0 0 0 0
```

MATRICES DINÁMICAS

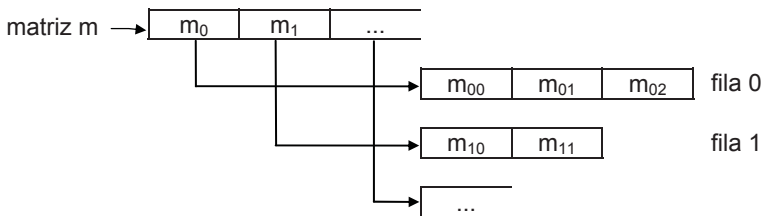
Hemos visto en los apartados anteriores que utilizando la técnica de asignar memoria dinámicamente, podremos decidir durante la ejecución cuántos elementos queremos que tenga nuestra matriz. Este tipo de matrices recibe el nombre de *matrices dinámicas* porque se crean durante la ejecución del programa. Igual que ocurría con las matrices primitivas estudiadas en el capítulo *Tipos estructurados de datos*, los elementos de una matriz dinámica pueden ser de cualquier tipo.

Para crear una matriz dinámica, además del operador **new**, la biblioteca de C++ proporciona contenedores como la plantilla **vector** que permite trabajar con matrices unidimensionales o multidimensionales de elementos de cualquier tipo, y la clase **string** especializada en el trabajo con cadenas de caracteres. Ambas fueron estudiadas y utilizadas, también, en el capítulo *Tipos estructurados de datos*.

Es aconsejable dar preferencia a la biblioteca estándar frente a otras bibliotecas y frente al código hecho a mano. Por lo tanto, utilice la plantilla **vector** en vez de las matrices primitivas (automáticas o dinámicas), dé preferencia a las opera-

ciones con **string** cuando trabaje con cadenas, y utilice **push_back** o **resize** sobre un contenedor de tipo **vector** en vez utilizar una matriz dinámica creada con **new**.

Como ejemplo vamos a realizar un programa que cree una matriz dinámica bidimensional de tipo **int** utilizando a plantilla **vector**. Una matriz de dos dimensiones puede ser rectangular (todas las filas tienen los mismos elementos) o dentada (las filas tienen distinto número de elementos). En cualquier caso, el proceso de crear dinámicamente una estructura de datos que funcione igual que una matriz de dos dimensiones se divide en dos partes:



- Crear una matriz de matrices vacía. Posteriormente, sus elementos se corresponderán con cada una de las filas de la supuesta matriz de dos dimensiones.

```
vector< vector<int> > m;
```

- Crear cada una de las filas. Inicialmente estarán vacías; después su número de elementos crecerá en lo necesario.

```
m.push_back(vector<int>(0)); // añadir una fila con 0 elementos
```

Como ejemplo, el programa que se muestra a continuación, crea dinámicamente una matriz de dos dimensiones, y finalmente la muestra.

```
// Matriz dinámica de dos dimensiones
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;
void mostrar(vector< vector<int> >&);
```

```
int main()
{
```

```
    vector< vector<int> > m; // crear una matriz de matrices vacía
```

```
    // Añadir elementos a la matriz m
```

```
    int f = 0, c = 0, dato = 0;
```

```
    while (!cin.eof())
```

```
    {
```

```
        m.push_back(vector<int>(0)); // añadir una fila
```

```

    cout << "Datos para la fila m[" << f << "]" (<eof> para finalizar)\n";
    c = 0;
    cout << "m[" << f << "]"[" << c << "] = ";
    cin >> dato;
    while (!cin.eof())
    {
        m[f].push_back(dato); // añadir un elemento a la fila f
        c++;
        cout << "m[" << f << "]"[" << c << "] = ";
        cin >> dato;
    }
    f++;
    cin.clear();
    cout << "\n(<Entrar> para otra fila, <eof> para finalizar)\n";
    string continuar;
    getline(cin, continuar); // hacer una pausa hasta pulsar
    // <Entrar> o <eof>
    cin.clear();

    // Mostrar la matriz m
    cout << endl;
    mostrar(m);
}

void mostrar(vector< vector<int> >& m)
{
    for (unsigned int f = 0; f < m.size(); f++)
    {
        for (unsigned int c = 0; c < m[f].size(); c++)
            cout << setw(8) << m[f][c];
        cout << endl;
    }
}

```

Ejecución del programa:

Datos para la fila m[0] (<eof> para finalizar)

m[0][0] = 123

m[0][1] = 3

m[0][2] = 45

m[0][3] = 76

...

(<Entrar> para otra fila, <eof> para finalizar)

Datos para la fila m[2] (<eof> para finalizar)

m[2][0] = 15

m[2][1] = 380

m[2][2] = 7

m[2][3] = 33

```
m[2][4] = ^Z
```

```
(<Entrar> para otra fila, <eof> para finalizar)
^Z
```

```
123      3      45      76      345
 34      56      7
15      380     7      33
```

En el ejemplo que acabamos de exponer, cada vez que se llama a **push_back**, la matriz *m* o *m[*ff*]* crece en un elemento, que se añade al final. Si en lugar de añadir un elemento al final quisiéramos añadirlo en una posición determinada, utilizaríamos el método **insert**. Ambas funciones incrementan implícitamente el tamaño del vector dado por el método **size**. El vector podrá crecer mientras haya memoria disponible para adquirir. El método **max_size** permite saber cuál es la longitud máxima posible que puede adquirir un vector.

Así mismo, cada vez que necesitemos reducir la matriz en un elemento, en el del final, invocaremos al método **pop_back**. Si en lugar del elemento del final quisiéramos eliminar uno de una posición específica, utilizaríamos el método **erase**. Al igual que el desbordamiento superior también se debe evitar el desbordamiento inferior; esto es:

```
if (m.size() > 0 ) m.pop_back(); // elimina el último elemento de m
```

También podemos modificar el tamaño de la matriz invocando al método **resize**. Por ejemplo, las siguientes líneas de código modifican el tamaño de la matriz *m* en *k* elementos iniciados con 0:

```
int nuevoTam = m.size() + k;
if (nuevoTam > 0) m.resize(nuevoTam);
mostrar(m);
```

Copiar un vector en otro es tan simple como realizar una asignación. El siguiente ejemplo copia el vector *v1* en *v2*. Ambos vectores tienen que ser del mismo tipo y *v2* será redimensionado automáticamente para ser igual que *v1*:

```
vector< vector<int> > v1, v2;
// ...
v2 = v1; // copiar el vector v1 en v2
```

También hemos visto que utilizando **push_back** o **insert** una matriz crece según se necesita. Pues bien, cuando conocemos de antemano que una matriz va a crecer, por lo menos, hasta un determinado tamaño, podemos anticiparnos y reservar el espacio necesario invocando al método **reserve**. Por ejemplo:

```
int main()
{
    vector<double> a; // a tiene 0 elementos
    // Reservar espacio para 10 elementos
    a.reserve(10); // a sigue teniendo 0 elementos
    // ...
}
```

La reserva de memoria con antelación tiene dos ventajas: una, no tener que estar adquiriendo la memoria cada vez que añadimos un nuevo elemento, y otra, que está garantizado que la matriz podrá almacenar al menos el número de elementos especificado (piense que otras aplicaciones que se ejecuten concurrentemente también pueden requerir memoria dinámicamente).

PUNTEROS A ESTRUCTURAS

Los punteros a estructuras se declaran igual que los punteros a otros tipos de datos. Para referirse a un miembro de una estructura apuntada por un puntero hay que utilizar el operador `->`.

Por ejemplo, el siguiente programa declara un puntero *hoy* a una estructura de tipo **struct** *fecha*, asigna memoria para la estructura, lee valores para cada miembro de la misma y, apoyándose en una función, escribe su contenido.

```
// Punteros a estructuras
#include <iostream>
using namespace std;

struct fecha
{
    unsigned int dd;
    unsigned int mm;
    unsigned int aa;
};

void escribir(fecha *f);

int main()
{
    struct fecha *hoy; // hoy es un puntero a una estructura

    try
    {
        // Asignación de memoria para la estructura
        hoy = new fecha;
        cout << "Introducir fecha (dd-mm-aa)\n";
        cout << "día: "; cin >> hoy->dd;
    }
}
```

```

    cout << "mes: "; cin >> hoy->mm;
    cout << "año: "; cin >> hoy->aa;
    escribir(hoy);
    delete hoy;
}
catch(bad_alloc e)
{
    cout << "Insuficiente espacio de memoria\n";
    return -1;
}
}

void escribir(fecha *f)
{
    cout << "Día " << f->dd << " del mes " << f->mm
        << " del año " << f->aa << endl;
}

```

Ejecución del programa:

```

Introducir fecha (dd-mm-aa)
día: 10
mes: 10
año: 2010
Día 10 del mes 10 del año 2010

```

Observe que el tipo *fecha* se ha declarado al principio, antes de cualquier función, lo que permite utilizarlo en cualquier parte.

Otro detalle importante es comprender que el simple hecho de declarar un puntero a una estructura no significa que dispongamos de la estructura; es necesario asignar al puntero un bloque de memoria del tamaño de la estructura donde se almacenarán los datos de la misma (este concepto es aplicable a cualquier tipo de objetos). Esto es, la declaración siguiente crea un puntero para apuntar a una estructura, pero no la estructura.

```
fecha *hoy = 0;
```



Por lo tanto sería un error ejecutar:

```
cin >> hoy->dd; cin >> hoy->mm; cin >> hoy->aa;
```

porque *dd*, *mm* y *aa*, ¿a qué estructura pertenecen? La respuesta es: a ninguna porque al puntero *hoy* no se le ha asignado una estructura. Observe el puntero *hoy*,

para mayor claridad está iniciado a cero, entonces no ha lugar a pensar que apunta a una estructura. Si hubiéramos hecho una declaración como la siguiente:

```
fecha f, *hoy = &f;
```

sí sería válido ejecutar las sentencias:

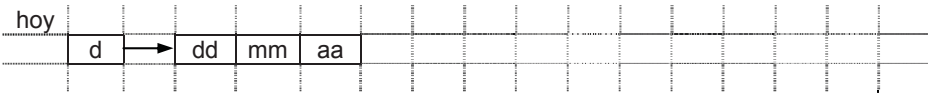
```
cin >> hoy->dd; cin >> hoy->mm; cin >> hoy->aa;
```

porque ahora *hoy* apunta a la estructura *f*. Pero si procedemos así, se preguntará, y con razón: ¿para qué queremos el puntero a la estructura? ¿Por qué no utilizar directamente la estructura *f* así?:

```
cin >> f.dd; cin >> f.mm; cin >> f.aa;
```

Esto evidencia que cuando declaramos un puntero a un objeto, casi siempre es porque el objeto va a ser creado durante la ejecución. Es decir:

```
fecha *hoy = 0;
hoy = new fecha;
```



Después de ejecutarse el operador **new**, *hoy* almacena la dirección *d* de un bloque de memoria reservado para almacenar una estructura del tipo *fecha*. Por lo tanto, ahora sí es correcto ejecutar las sentencias:

```
cin >> hoy->dd; cin >> hoy->mm; cin >> hoy->aa;
```

Si *hoy* es un puntero a una estructura de tipo *fecha*, **hoy* es esa estructura. Por lo tanto, las sentencias anteriores son equivalentes a las siguientes:

```
cin >> (*hoy).dd; cin >> (*hoy).mm; cin >> (*hoy).aa;
```

Si en lugar de trabajar con estructuras tenemos que trabajar con uniones, no hay diferencias; los punteros a uniones se manipulan exactamente igual que los punteros a estructuras.

PUNTEROS COMO PARÁMETROS EN FUNCIONES

Volvamos al ejemplo anterior y fijémonos en la función *escribir*. Tiene un parámetro que es un puntero.

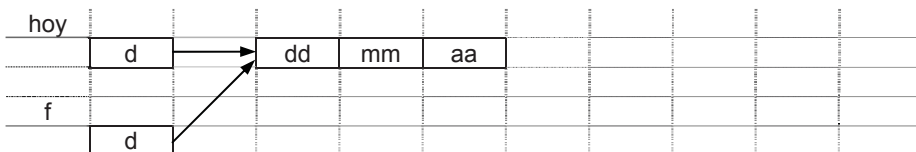
```
int main()
{
    fecha *hoy = 0; // hoy es un puntero a una estructura
    // ...
    escribir(hoy);
    // ...
}
```

```
void escribir(fecha *f)
{
    cout << "Día " << f->dd << " del mes " << f->mm
         << " del año " << f->aa << endl;
}
```

Cuando **main** invoca a la función *escribir*, ¿qué ocurre?

1. El parámetro *f* almacena el valor del argumento *hoy*; esto es, se realiza la operación $f = hoy$.
2. Ahora *f* y *hoy* apuntan a la misma estructura de datos. Por lo tanto, la función *escribir* utilizando el puntero *f* puede acceder a los mismos datos miembro de la estructura que la función **main** utilizando el puntero *hoy*.

¿Cómo se ha pasado el argumento *hoy*? Evidentemente por valor. Según hemos dicho, en la llamada a la función *escribir* se realizó la operación $f = hoy$.



¿Cómo se ha pasado la estructura? Evidentemente por referencia, porque *escribir* no recibe una copia de la estructura, sino la dirección *hoy* donde está ubicada la misma.

Por lo tanto, si *escribir* cambiara el valor de *f*, *hoy* no se modificaría, pero si cambiara algún miembro de la estructura, esos cambios también serían vistos desde **main**. Piénselo sobre la figura anterior. Si quisiéramos que *hoy* fuera modificado por una función, habría que pasar este argumento por referencia, lo que implica definir en la función el parámetro formal correspondiente como un puntero a un puntero, o bien como una referencia a un puntero.

Pongamos otro ejemplo para aclarar este último aspecto. Vamos a escribir una función *asigem* a la que **main** debe de invocar para reservar memoria para una estructura. Analicemos esta primera versión:

```

int main()
{
    fecha *hoy = 0; // hoy es un puntero a una estructura

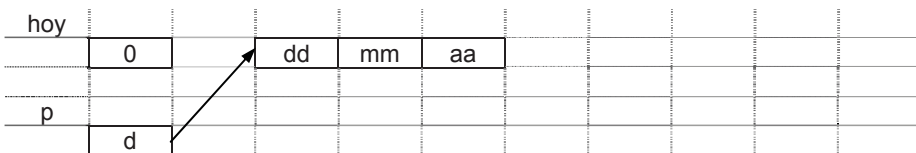
    // Asignación de memoria para la estructura
    asigmem(hoy);
    if (hoy == 0) return -1;
    // ...
}

void asigmem(fecha *p)
{
    try
    {
        p = new fecha;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        return;
    }
}

```

¿Cómo se pasa el argumento *hoy*? Evidentemente por valor, porque cuando se ejecuta la llamada a la función *asigmem* se realiza la operación $p = hoy$. Como *hoy* vale cero, *p* inicialmente también valdrá cero.

Se ejecuta la función *asigmem* y *p* toma un nuevo valor: la dirección *d* del bloque de memoria reservado para una estructura de tipo *fecha*. ¿Ha cambiado *hoy* en el mismo valor? Según el análisis realizado anteriormente, no; *hoy* sigue valiendo cero, por lo tanto, el programa no funciona. Además, cuando la función *asigmem* finalice, la variable local *p* será destruida y quedará un bloque de memoria sin referenciar y sin liberar (se ha generado una laguna de memoria).



Hagamos otra versión en la que se pase el puntero *hoy* por referencia, con la intención de que *asigmem* pueda acceder a su contenido y modificarlo almacenando en él la dirección del bloque de memoria por ella reservado.

```

// Punteros como parámetros
#include <iostream>
using namespace std;

```



```
struct fecha
{
    unsigned int dd;
    unsigned int mm;
    unsigned int aa;
};

void escribir(fecha *);
void asigmem(fecha *&); // el parámetro es una referencia a un
                        // puntero a fecha

int main()
{
    fecha *hoy = 0; // hoy es un puntero a una estructura
    // Asignación de memoria para la estructura
    try
    {
        asigmem(hoy);
        cout << "Introducir fecha (dd-mm-aa)\n";
        cout << "día: "; cin >> hoy->dd;
        cout << "mes: "; cin >> hoy->mm;
        cout << "año: "; cin >> hoy->aa;
        escribir(hoy);
        delete hoy;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        return -1;
    }
}

void escribir(fecha *f)
{
    cout << "Día " << f->dd << " del mes " << f->mm
        << " del año " << f->aa << endl;
}

void asigmem(fecha*& p)
{
    try
    {
        p = new fecha;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        return;
    }
}
```

Observando la función *asigem*, vemos que ahora *p* se ha definido como una referencia al puntero *hoy*; esto es, *p* es sinónimo de *hoy*. Entonces, lo que se haga con *p*, en realidad se está haciendo con *hoy*, variable en la cual se almacena la dirección de memoria devuelta por **new**.

Otra versión de la función *asigem* anterior puede ser:

```
void asigem(fecha **pp);
```

Y otra versión más de la función *asigem* anterior puede ser una que no tenga parámetros y devuelva la dirección del bloque de memoria reservado para una estructura de tipo *fecha*:

```
fecha *asigem()
{
    fecha *p;
    try
    {
        p = new fecha;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        return 0;
    }
    return p;
}
```

La llamada a esta función sería ahora así:

```
hoy = asigem();
```

DECLARACIONES COMPLEJAS

Entendemos por declaración compleja un identificador calificado por más de un operador (matriz: [], puntero: *, referencia: &, o función: ()). Se pueden aplicar varias combinaciones con estos operadores sobre un identificador; sin embargo, los elementos de una matriz no pueden ser funciones y una función no puede devolver como resultado una matriz o una función.

Para interpretar estas declaraciones, hay que saber que los corchetes y paréntesis (operadores a la derecha del identificador) tienen prioridad sobre los asteriscos y ampersands (operadores a la izquierda del identificador). Los paréntesis y corchetes tienen la misma prioridad y se evalúan de izquierda a derecha. Como último paso se aplica el tipo especificado. Utilizando paréntesis, podemos cambiar

el orden de prioridades. Las expresiones entre paréntesis se evalúan primero, de más internas a más externas.

Una forma sencilla de interpretar declaraciones complejas es leerlas desde dentro hacia afuera, siguiendo los pasos indicados a continuación:

1. Comenzar con el identificador y mirar si hacia la derecha hay corchetes o paréntesis.
2. Interpretar esos corchetes o paréntesis y mirar si hacia la izquierda del identificador hay asteriscos o ampersands.
3. Dentro de cada nivel de paréntesis, de más internos a más externos, aplicar las reglas 1 y 2.

El siguiente ejemplo clarifica lo expuesto. En él se han enumerado el identificador *var*, los calificadores `[]`, `()` y `*`, y el tipo **char**, en el orden de interpretación resultado de aplicar las reglas anteriores.

```
char *(*(*var)())[10]
  ↑   ↑ ↑ ↑ ↑   ↑   ↑
  7   6 4 2 1   3   5
```

La lectura que se hace al interpretar la declaración anterior es:

1. El identificador *var* es declarado como
2. un puntero a
3. una función que devuelve
4. un puntero a
5. una matriz de 10 elementos, los cuales son
6. punteros a
7. objetos de tipo **char**.

EJERCICIOS RESUELTOS

1. Se quiere escribir un programa para manipular polinomios. Para ello, vamos a utilizar una estructura de datos como la siguiente:

```
typedef struct
{
    int grado;           // grado del polinomio
    vector<float> coef;  // coeficientes del polinomio
} tpolinomio;
```

El miembro *grado* es un valor mayor que 0 que especifica el grado del polinomio. El miembro *coef* es una matriz cuyos elementos contienen los coeficientes del polinomio. El número de elementos de la matriz es el número de coeficientes del polinomio y depende del grado de éste. Por ejemplo, sea el polinomio: $x^5+5x^3-7x^2+4$.

Como el grado del polinomio es 5, la matriz de los coeficientes tendrá seis elementos cuyos valores serán: 1, 0, 5, -7, 0 y 4.

Se pide:

- a) Escribir una función *LeerPol* que lea a través del teclado un polinomio y lo almacene en una estructura del tipo *tpolinomio* anteriormente descrita. Para el polinomio que hemos puesto como ejemplo anteriormente, la entrada de datos se efectuaría así:

```
Grado del polinomio: 5
Coeficientes de mayor a menor grado: 1 0 5 -7 0 4
```

El prototipo de esta función será el siguiente:

```
void LeerPol(tpolinomio& pol);
```

- b) Escribir una función *VisualizarPol* que visualice en pantalla un polinomio. Por ejemplo, el polinomio puesto como ejemplo anteriormente sería visualizado así:

```
+1x^5 +5x^3 -7x^2 +4
```

El prototipo de la función será el siguiente:

```
void VisualizarPol(tpolinomio& pol);
```

El parámetro *pol* es una estructura que especifica el polinomio a visualizar.

- c) Escribir una función *SumarPols* que devuelva como resultado la suma de dos polinomios. El prototipo de esta función será:

```
tpolinomio SumarPols(tpolinomio& polA, tpolinomio& polB);
```

Los parámetros *polA* y *polB* son estructuras que especifican los polinomios a sumar.

- d) Utilizando las funciones anteriores, escribir un programa que lea dos polinomios y visualice en pantalla su suma.

El programa completo se muestra a continuación.

```
// Polinomios
#include <iostream>
#include <vector>
using namespace std;

typedef struct
{
    int grado;           // grado del polinomio
    vector<float> coef;  // coeficientes del polinomio
} tpolinomio;

void LeerPol(tpolinomio& pol)
{
    cout << "Grado del polinomio: ";
    cin >> pol.grado;
    // Reservar espacio en el vector para los coeficientes
    pol.coef.reserve(pol.grado + 1);

    // Leer los coeficientes de mayor a menor grado
    cout << "Coeficientes de mayor a menor grado: ";
    float val;
    for (int i = pol.grado; i >= 0; i--)
    {
        cin >> val;
        // Insertar todos los coeficientes por el principio
        pol.coef.insert(pol.coef.begin(), val);
    }
}

void VisualizarPol(tpolinomio& pol)
{
    int i;
    // Escribir los términos de pol de mayor a menor grado
    for (i = pol.grado; i > 0; i--)
        if (pol.coef[i])
            cout << showpos << pol.coef[i] << "x^" << i << ' ';
    // Escribir el término independiente
    if (pol.coef[i]) cout << showpos << pol.coef[i] << endl;
}

tpolinomio SumarPols(tpolinomio& polA, tpolinomio& polB)
{
    tpolinomio polresu, polaux;
    // Hacer que polA sea el de mayor grado
    if (polA.grado < polB.grado)
    {
        polaux = polA;
        polA = polB;
    }
}
```

```
    polB = polaux;
}
// El polinomio resultante tendrá como grado, el mayor
polresu.grado = polA.grado;
// Reservar espacio para los coeficientes de polresu
polresu.coef.reserve(polresu.grado + 1);

// Sumar polB con los coeficientes correspondientes de polA
int i;
for (i = 0; i <= polB.grado; i++)
    polresu.coef.push_back(polB.coef[i] + polA.coef[i]);

// A partir del valor actual de i, copiar
// los coeficientes restantes de polA
for (; i <= polA.grado; i++)
    polresu.coef.push_back(polA.coef[i]);

return polresu;
}

int main()
{
    tpolinomio polA, polB, polR;

    LeerPol(polA);
    if (polA.coef.size() != 0)
    {
        LeerPol(polB);
        if (polB.coef.size() != 0)
        {
            polR = SumarPols(polA, polB);
            if (polR.coef.size() != 0)
            {
                VisualizarPol(polR);
            }
        }
    }

    if ((polA.coef.size() == 0) || (polB.coef.size() == 0) ||
        (polR.coef.size() == 0))
        cout << "Error: polinomio vacío\n";
}
```

Ejecución del programa:

```
Grado del polinomio: 5
Coeficientes de mayor a menor grado: 1 0 5 -7 0 4
Grado del polinomio: 3
Coeficientes de mayor a menor grado: -3 7 1 -3
+1x^5 +2x^3 +1x^1 +1
```

2. Queremos generar un diccionario inverso. Estos diccionarios se caracterizan por presentar las palabras en orden alfabético ascendente pero observando las palabras desde su último carácter hasta el primero (por ejemplo: hola → aloh). En la tabla siguiente podemos ver un ejemplo de este tipo de ordenación:

DICCIONARIO NORMAL	DICCIONARIO INVERSO
adiós	hola
camión	rosa
geranio	camión
hola	geranio
rosa	tractor
tractor	adiós

Una aplicación de este curioso diccionario es buscar palabras que rimen. Para escribir un programa que genere un diccionario de este tipo, se pide:

- a) Escribir la función *Comparar* cuyo prototipo es el siguiente:

```
int Comparar(const char *cad1, const char *cad2);
```

Esta función comparará *cadena1* y *cadena2*, pero observando las palabras desde su último carácter hasta el primero. La función devolverá los siguientes resultados:

- > 0 Si *cadena1* está alfabéticamente después que *cadena2*.
- 0 Si *cadena1* y *cadena2* son iguales.
- < 0 Si *cadena1* está alfabéticamente antes que *cadena2*.

- b) Escribir la función *OrdenarPalabras* con el prototipo que se indica a continuación, para ordenar las palabras de la matriz *palabra* en orden alfabético ascendente:

```
void OrdenarPalabras(vector<string>& palabra);
```

El parámetro *palabra* es la matriz que contiene las palabras. Para ordenar las palabras se empleará el *método de inserción* (para detalles sobre este método de ordenación, vea el capítulo de *Algoritmos*).

- c) Escribir un programa que lea palabras desde la entrada estándar y las almacene en una matriz dinámica de cadenas de caracteres, y tras ordenarlas utilizando las funciones anteriores, las visualice en la salida estándar. Para ello escriba, además de las funciones anteriores, las siguientes funciones:

```
void LeerPalabras(vector<string>& palabra);
```

```
void VisualizarPalabras(vector<string>& palabra);
```

El programa completo se muestra a continuación.

```
// Diccionario inverso
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void LeerPalabras(vector<string>&);
int Comparar(const char *cad1, const char *cad2);
void OrdenarPalabras(vector<string>& palabra);
void VisualizarPalabras(vector<string>&);

int main()
{
    // Matriz de cadenas de caracteres inicialmente vacía
    vector<string> palabra;
    // Reservar un espacio inicial para 100 objetos string
    palabra.reserve(100);
    // Operaciones
    LeerPalabras(palabra);
    OrdenarPalabras(palabra);
    cout << '\n';
    VisualizarPalabras(palabra);
}

void LeerPalabras(vector<string>& palabra)
{
    cout << "Escriba las palabras que desea introducir.\n";
    cout << "Puede finalizar con <eof>\n";
    int fila = 0;
    string pal; // una palabra
    cout << "palabra[" << fila++ << "]: ";
    getline(cin, pal);
    while (!cin.eof()) // si se pulsó [Ctrl][z], salir del bucle
    {
        palabra.push_back(pal); // añadir una palabra
        cout << "palabra[" << fila++ << "]: ";
        getline(cin, pal); // leer otra palabra
    }
    cin.clear(); // desactivar el indicador de eof
}

int Comparar(const char *cad1, const char *cad2)
{
    int i, j;

    i = strlen(cad1) - 1;
```



```

j = strlen(cad2) - 1;
// Comparar las palabras de atrás hacia adelante
while( i > 0 && j > 0 )
{
    if ( cad1[i] != cad2[j] )
        return (cad1[i] - cad2[j]);
    i--;
    j--;
}
return (cad1[i] == cad2[j]) ? i - j : cad1[i] - cad2[j];
// (i - j) para parejas como "centrar" y "entrar"
}

void OrdenarPalabras(vector<string>& palabra)
{
    string aux;
    int i = 0, k = 0;
    size_t npal = palabra.size();

    // Ordenar: método de inserción
    for (i = 1; i < npal; i++)
    {
        aux = palabra[i];
        k = i - 1;
        while ((k >= 0) && (Comparar(aux.c_str(), palabra[k].c_str()) < 0))
        {
            palabra[k+1] = palabra[k];
            k--;
        }
        palabra[k+1] = aux;
    }
}

void VisualizarPalabras(vector<string>& palabra)
{
    vector<string>::iterator pal; // pal es un iterador
    for (pal = palabra.begin(); pal != palabra.end(); ++pal)
        cout << *pal << '\n'; // mostrar una palabra
}

```

Ejecución del programa:

Escriba las palabras que desea introducir.

Puede finalizar con <eof>

palabra[0]: adiós

palabra[1]: camión

palabra[2]: geranio

palabra[3]: hola

palabra[4]: rosa

palabra[5]: tractor

```
palabra[6]: ^Z
```

```
hola  
rosa  
camión  
geranio  
tractor  
adiós
```

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

1) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int *p = 0, a = 0;  
    *p = 101;  
    a = *p;  
    cout << a << endl;  
}
```

- a) 0.
- b) 101.
- c) No se puede ejecutar porque hay errores durante la compilación.
- d) No se puede ejecutar porque hay errores durante la ejecución.

2) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>  
using namespace std;  
int main()  
{  
    double d = 1024.77;  
    int *p, a = 0;  
    p = &d;  
    a = *p;  
    cout << a << endl;  
}
```

- a) Imprime 1024.
- b) Imprime basura (valor no predecible).
- c) Produce un error durante la compilación.
- d) Produce un error durante la ejecución.

3) ¿Cuál es el resultado del siguiente programa?

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

int main()
{
    int a[5] = { 10, 20, 30, 40, 50 };
    int *p = a;
    cout << *(p + 2) << endl;
}
```

- a) Imprime basura (valor no predecible).
- b) Imprime 30.
- c) Produce un error durante la compilación.
- d) Ninguno de los anteriores.

4) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
using namespace std;
int main()
{
    int a[5] = { 10, 20, 30, 40, 50 };
    cout << *a++ << endl;
}
```

- a) La dirección de a[1].
- b) 10.
- c) El programa produce un error durante la ejecución.
- d) El programa produce un error durante la compilación.

5) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
using namespace std;
int main()
{
    int a[5] = {10, 20, 30, 40, 50};
    int& rElemento = a[0];
    int *p = &rElemento;
    rElemento++;
    cout << *p << ' ';
}
```

- a) 11.
- b) 10.
- c) El programa produce un error durante la ejecución.
- d) El programa produce un error durante la compilación.

6) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
using namespace std;
int main()
{
    int a[2][3] = { 10, 20, 30, 40, 50, 60 };
    int **p = a;
    cout << p[1][1] << endl;
}
```

- a) 50.
- b) 20.
- c) El programa produce un error durante la compilación.
- d) El programa produce un error durante la ejecución.

7) ¿Cuál es el resultado del siguiente programa?

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

int main()
{
    int a[2][3] = { 10, 20, 30, 40, 50, 60 };
    int *p[3];
    p[0] = a[0], p[1] = a[1];
    cout << p[1][1] << endl;
}
```

- a) 50.
- b) 20.
- c) El programa produce un error durante la compilación.
- d) El programa produce un error durante la ejecución.

8) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
using namespace std;
int main()
{
    char *c[] = { "abc", "def", "ghi" };
    cout << c[1] << endl;
}
```

- a) abc.
- b) def.
- c) Produce un error durante la compilación.
- d) Produce un error durante la ejecución.

9) ¿Cuál es el resultado del siguiente programa?

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

void asigmem(int*&, int);
int main()
{
    int *p = 0;
    asigmem(p, 3);
    p[0] = 10, p[1] = 20, p[2] = 30;
    cout << p[1] << endl;
    delete [] p;
}
void asigmem(int*& p, int t)
{
    p = new int[t];
}
```

- a) 20.
- b) Imprime basura (valor indeterminado).
- c) Produce un error durante la compilación.
- d) Produce un error durante la ejecución.

10) ¿Cuál es el resultado del siguiente programa?

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

union tdato
{
    int a;
    float b;
};

int main()
{
    tdato *s = 0;
    s->a = 10;
    cout << s->a << ' ' << s->b << endl;
}
```

- a) Imprime 10 y un valor indeterminado.
- b) 10 0.
- c) Produce un error durante la ejecución.
- d) Imprime basura (valor indeterminado).

2. Realizar un programa que permita utilizar el terminal como un diccionario inglés-español; esto es, al introducir una palabra en inglés, se escribirá la correspondien-

te palabra en español. Como ejemplo, supongamos que introducimos las siguientes parejas de palabras:

book	libro
green	verde
mouse	ratón

Una vez finalizada la introducción de la lista, pasamos al modo traducción, de forma que si tecleamos *green*, la respuesta ha de ser *verde*. Si la palabra no se encuentra, se emitirá un mensaje que lo indique.

El programa constará al menos de dos funciones:

- a) *CrearDiccionario*. Esta función creará el diccionario.
 - b) *Traducir*. Esta función realizará la labor de traducción.
3. Un cuadrado mágico se compone de números enteros comprendidos entre 1 y n^2 , donde n es un número impar que indica el orden de la matriz cuadrada que contiene los números que forman dicho cuadrado mágico. La matriz que forma este cuadrado mágico cumple que la suma de los valores que componen cada fila, cada columna y cada diagonal es la misma. Por ejemplo, un cuadrado mágico de orden 3, $n = 3$, implica una matriz de 3 por 3. Por lo tanto, los valores de la matriz estarán comprendidos entre 1 y 9 y dispuestos de la siguiente forma:

8	1	6
3	5	7
4	9	2

Realizar un programa que visualice un cuadrado mágico de orden impar n . El programa verificará que n es impar y está comprendido entre 3 y 15.

Una forma de construirlo 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 tener en cuenta que el cuadrado se cierra sobre sí mismo; esto es, la línea encima de la primera es la última y la columna a la derecha de la última es la primera. Siguiendo esta regla, cuando el número caiga en una casilla ocupada, se elige la casilla situada debajo del último número situado.

Se deberán realizar al menos las siguientes funciones:

- a) *Es_impar*. Esta función verificará si n es impar.
 - b) *Cuadrado_mágico*. Esta función construirá el cuadrado mágico.
4. Una empresa dedicada a la venta de electrodomésticos y a su posterior mantenimiento desea tener una aplicación que automatice todos sus procesos de gestión.

Esto supone tener la información de todos los clientes que compran electrodomésticos junto con los contratos de mantenimiento (esta última información, lógicamente, sólo estará disponible en los casos que el cliente contrate un seguro de mantenimiento para los electrodomésticos adquiridos) almacenada en una matriz dinámica de estructuras.

Cada cliente podrá asegurar o no el electrodoméstico comprado y cada electrodoméstico asegurado dará lugar a un contrato de mantenimiento.

La estructura *tcon* almacena la información de cada electrodoméstico comprado y asegurado:

```
struct tcon
{
    string Descripcion;    // Descripción del electrodoméstico
    string NumSerie;      // Número de serie del aparato
    double ValorCompra;   // Valor del electrodoméstico
    string NumContrato;   // Número del contrato
    double ImpContrato;   // Importe del contrato de mantenimiento
};
```

Para almacenar los datos de los clientes que han comprado electrodomésticos aunque no hayan asegurado ninguno, se define una estructura *tcliente* así:

```
struct tcliente
{
    string Nombre;        // Nombre del cliente
    string Apellidos;    // Apellidos del cliente
    string Direccion;    // Dirección del cliente
    string Codigo;       // Código del cliente
    vector<tcon> Contrato; // vector de estructuras tcon
};
```

El vector *Contrato* se define para almacenar los contratos de mantenimiento suscritos por cada cliente.

La matriz dinámica de estructuras con la información de los clientes estará referenciada por la variable *cliente* definida a continuación:

```
vector<tcliente> cliente;
```

Partiendo de las declaraciones anteriores y suponiendo que existe una matriz *cliente* correctamente iniciada, implemente las siguientes funciones:

Valor retornado:

Un 0 si el importe de la reparación excede el 25% de la compra, un 1 si se autoriza y un 2 si el número de serie no existe.

- d) Función *BuscarPosicion*. Permite encontrar la posición de un determinado cliente en la matriz dinámica de estructuras. El prototipo para esta función es el siguiente:

```
int BuscarPosicion(vector<tcliente>& cliente, string codigo);
```

Parámetros:

cliente matriz de estructuras *tcliente*.
codigo código del cliente.

Valor retornado:

Un entero que indica la posición que ocupa el cliente en la matriz, o -1 si el cliente no se encuentra.

- e) Función *Listar*. Mostrará en pantalla un listado de todos los datos de los clientes. En el caso de que un cliente no tenga ningún contrato mostrará un mensaje indicándolo.

```
void Listar(const vector<tcliente>& cliente);
```

Parámetros:

pcliente puntero a la matriz de estructuras *tcliente*.

Valor retornado: ninguno.

- f) Función *menu*. Mostrará en pantalla un menú como el siguiente:

```
1.- Añadir un nuevo cliente
2.- Dar de alta un contrato
3.- Visualizar todos los datos de todos los clientes
4.- ¿Se autoriza la reparación?
5.- Posición de un cliente
6.- Salir
Introduzca la opción:
```

El prototipo para esta función es el siguiente:

```
int menu();
```

Parámetros: ninguno.

Valor retornado: un entero correspondiente a la opción elegida.

- g) Función *LeerInt*. Leerá un valor de tipo **int** desechando cualquier entrada que sea incorrecta. El prototipo para esta función es el siguiente:

```
int LeerInt();
```

Parámetros: ninguno.

Valor retornado: el entero leído.

- h) Función *LeerDouble*. Leerá un valor de tipo **double** desechando cualquier entrada que sea incorrecta. El prototipo para esta función es el siguiente:

```
double LeerDouble();
```

Parámetros: ninguno.

Valor retornado: el valor real leído.

- i) Función **main**. Visualizará el menú y permitirá realizar las operaciones especificadas.

MÁS SOBRE FUNCIONES

En los capítulos anteriores hemos aprendido lo que es un programa, cómo escribirlo y qué hacer para que el ordenador lo ejecute y muestre los resultados conseguidos; aprendimos acerca de los elementos que aporta C++; analizamos cómo era la estructura de un programa C++; aprendimos a leer datos desde el teclado y a visualizar resultados sobre el monitor; estudiamos las estructuras de control; trabajamos con matrices y aprendimos a utilizar punteros.

Así mismo, en el capítulo 3 se introdujo el concepto de función como unidad independiente de un programa C++. Desde entonces sabemos que todo programa C++ está formado además de por la función **main**, que es el punto de entrada y de salida del programa, por otras funciones, las cuales se comunican entre sí pasándose argumentos siempre que sean requeridos. Todo ello lo hemos venido aplicando en los ejercicios realizados en los capítulos estudiados hasta aquí.

Por eso, en este capítulo vamos a centrarnos en cuestiones más específicas, como estudiar la problemática que se nos puede presentar, bien al pasar matrices, estructuras y punteros como argumentos a funciones, o bien cuando una función devuelve estos tipos de datos. También abordaremos el paso de argumentos a través de la línea de órdenes, las funciones recursivas, los punteros a funciones, funciones con parámetros por omisión, funciones en línea, funciones sobrecargadas y operadores sobrecargados.

PASAR UNA MATRIZ COMO ARGUMENTO A UNA FUNCIÓN

Atendiendo al instante en el que se hace la reserva de memoria para una matriz, la denominaremos automática (o estática) cuando sea el compilador el que haga dicha reserva y dinámica cuando la reserva se haga durante la ejecución del pro-

grama utilizando el operador **new**, o bien contenedores como **vector** o **map**. Recordar que el nombre de una matriz automática es una constante.

Matrices automáticas

Ya hemos dicho en numerosas ocasiones que el nombre de una matriz es la dirección de comienzo de dicha matriz y también hemos visto que cuando pasamos una matriz a una función el argumento especificado en la llamada es exclusivamente el nombre de la matriz. Esto significa que lo que se pasa es la dirección de la matriz y, por lo tanto, el parámetro formal correspondiente en la definición de la función debe ser una matriz del mismo tipo, el cual, después de la llamada, quedará iniciado con esa dirección. Por eso se dice que las matrices son siempre pasadas por referencia. Si fueran pasadas por valor, se pasaría una copia de todos sus elementos con el consiguiente coste de recursos y de tiempo.

Por consiguiente, cuando se pasa una matriz a una función, lo que ésta conoce es el lugar de la memoria donde está ubicada esa matriz. De esta forma, tanto la función llamante como la llamada trabajan sobre el mismo espacio de memoria; en otras palabras, sobre la misma matriz. Cualquier cambio que haga una de ellas sobre la matriz será visto por la otra. El siguiente ejemplo muestra con detalle lo que acabamos de explicar:

```
// ...
int main() // función principal
{
    static tficha biblioteca[N]; // matriz de estructuras
    int n = 0; // número actual de elementos con datos en la matriz
    cout << "Introducir datos.\n";
    n = leer(biblioteca, N);
    cout << "Listado de libros y revistas\n";
    escribir(biblioteca, n); // listar todos los libros y revistas
}

int leer(tficha bibli[], int NMAX)
{
    // ...
}

void escribir(tficha bibli[], int n)
{
    // ...
}
```

En este ejemplo, el primer parámetro de la función *leer* y de la función *escribir* es una matriz de una dimensión. Recuerde que cuando se declara una matriz unidimensional como parámetro de una función, no se requiere que se especifique

su dimensión, simplemente porque la matriz ya existe y lo único que se necesita es declarar una variable del tipo de la matriz, en el ejemplo del tipo *tficha* [], para almacenar la dirección de la misma. Si la matriz es multidimensional, entonces no se requiere que se especifique la primera dimensión, pero sí las restantes, como puede ver a continuación; el razonamiento es el mismo: como la matriz existe, lo único que se requiere es una variable del tipo de la matriz, en el ejemplo **float** [][COLS], para almacenar la dirección de la misma.

```
// ...
int main()
{
    static float a[FILAS][COLS], c[FILAS][COLS];
    int fila = 0, col = 0;

    // Leer datos para la matriz a
    for (fila = 0; fila < FILAS; fila++)
    {
        for (col = 0; col < COLS; col++)
        {
            cout << "a[" << fila << "]" << col << "] = ";
            cin >> a[fila][col];
        }
    }

    // Copiar la matriz a en c
    CopiarMatriz(c, a);
    // ...

void CopiarMatriz( float destino[][COLS], float origen[][COLS] )
{
    // ...
}
```

En este otro ejemplo, los dos parámetros de la función *CopiarMatriz* son matrices de dos dimensiones. El tener que especificar la segunda dimensión hace que la función dependa de ese valor externo, lo que supone declarar esa constante cuando utilicemos esta función en otros programas. Esto podría solucionarse con un fichero de cabecera en el que se incluyera tanto el prototipo de la función como la definición de la constante.

Por otra parte, si el nombre de una matriz representa una dirección de un nivel de indirección, ¿podríamos almacenar esa dirección en un puntero también con un nivel de indirección? Sí, si se define adecuadamente el puntero. Para matrices de una dimensión, como el nombre de una matriz direcciona el primer elemento de dicha matriz, bastaría con definir un puntero al tipo del elemento. Como ejemplo, observar el parámetro *bibli* de la función *leer* que anteriormente fue declarado

como una matriz, y ahora es declarado como un puntero; el comportamiento es el mismo en ambos casos.

```
int leer(tficha *bibli, int NMAX)
{
    // ...
}
```

En cambio, no podríamos hacer exactamente lo mismo con *CopiarMatriz*, porque, según aprendimos en el capítulo anterior, el nombre de una matriz de dos dimensiones también define un nivel de indirección, pero no a un elemento, sino a una fila. Según esto, la función *CopiarMatriz* podría ser así:

```
void CopiarMatriz(float (*destino)[COLS], float (*origen)[COLS] )
{
    int fila = 0, col = 0;

    for (fila = 0; fila < FILAS; fila++)
    {
        for (col = 0; col < COLS; col++)
            destino[fila][col] = origen[fila][col];
    }
}
```

No obstante, si sustituimos el acceso indexado por un acceso a través de la dirección de cada elemento, calculada mediante aritmética de punteros a partir de la dirección de la matriz, el problema también queda resuelto. El ejemplo siguiente muestra esta otra forma de proceder:

```
void CopiarMatriz( float *destino, float *origen )
{
    int fila = 0, col = 0;

    for (fila = 0; fila < FILAS; fila++)
    {
        for (col = 0; col < COLS; col++)
            *(destino + (fila*COLS) + col) = *(origen + (fila*COLS) + col);
    }
}
```

Matrices dinámicas y contenedores

Una matriz construida con el operador **new** está referenciada por una variable de tipo puntero con tantos niveles de indirección como dimensiones tenga la matriz. Por lo tanto, pasar una matriz dinámica como argumento a una función simple-

mente requiere que el parámetro formal correspondiente sea un puntero del mismo tipo. Por ejemplo:

```
void Visualizar(double *, int);

int main()
{
    int elems = 6; // número de elementos de la matriz
    // Crear una matriz unidimensional dinámicamente
    double *m = new (nothrow) double[elems];
    // ...
    // Visualizar la matriz
    Visualizar(m, elems);
    // ...

    delete [] m;
}

void Visualizar(double *x, int elems)
{
    // ...
}
```

En cambio, si utilizamos un contenedor de tipo **vector**, no tendremos que preocuparnos de la gestión de la memoria y, por lo tanto, evitaremos el uso de punteros. Por ejemplo:

```
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;

void Visualizar(vector<double>);

int main()
{
    // Crear una matriz unidimensional dinámicamente
    vector<double> m;
    int n = 10;
    m.reserve(n); // reservar espacio para n elementos
    // Crear los elementos y asignar datos
    for (int i = 0; i < n; i++)
        m.push_back(i*2);
    // Visualizar la matriz
    Visualizar(m);
}

void Visualizar(vector<double> x)
{
```

```

    for (int i = 0; i < x.size(); i++)
        cout << setw(5) << x[i];
    cout << endl;
}

```

En este programa, la función **main** crea una matriz *m* inicialmente vacía, después reserva memoria para *n* elementos y a continuación construye los elementos. Finalmente, **main** invoca a una función *Visualizar* para mostrar el contenido de la matriz. A diferencia de lo que ocurría con las matrices automáticas, un vector se pasa por valor; para pasarlo por referencia hay que especificarlo. Por ejemplo:

```
void Visualizar(vector<double>&);
```

```

int main()
{
    // Crear una matriz unidimensional dinámicamente
    vector<double> m;
    // ...
    Visualizar(m);
}

```

```

void Visualizar(vector<double>& x) // parámetro pasado por referencia
{
    // ...
}

```

Hay dos razones para pasar un vector por referencia: una, que evitamos hacer un duplicado del vector, lo que supone menor tiempo de ejecución, y otra, que la función llamada puede hacer cambios sobre él, siempre que sea necesario. No obstante, podemos pasar el vector por referencia y evitar que la función llamada, por cualquier circunstancia, pueda modificar el vector original; basta para ello con declarar **const** el vector. Por ejemplo:

```

void Visualizar(const vector<double>& x)
{
    // ...
}

```

PASAR UN PUNTERO COMO ARGUMENTO A UNA FUNCIÓN

Un puntero, igual que otros tipos de variables, puede ser pasado por valor o por referencia. *Por valor* significa que el valor almacenado (una dirección) en el parámetro actual (argumento especificado en la llamada a la función) se copia en el parámetro formal correspondiente (parámetro declarado en la cabecera de la función); si ahora modificamos el valor de este parámetro formal, el parámetro actual correspondiente no se verá afectado. *Por referencia* lo que se copia no es la

dirección almacenada en el parámetro actual, sino la dirección en la que se localiza ese parámetro actual.

Analicemos la versión simplificada del siguiente programa. Cuando **main** invoca a la función *AsignarMem2D*, ¿cómo se pasa el argumento *m*? Evidentemente por valor, porque cuando se ejecuta la llamada se realiza la operación $x = m$. Como *m* vale 0, *x* inicialmente también valdrá 0.

```
int main()
{
    double **m = 0;
    // ...
    AsignarMem2D(m, filas, cols);
    Visualizar(m, filas, cols);
    LiberarMem2D(m, filas);
}

void AsignarMem2D(double **x, int filas, int cols)
{
    x = new (nothrow) double *[filas];
    // ...
}

void LiberarMem2D(double **x, int filas)
{
    // ...
}

void Visualizar(double **x, int filas, int cols)
{
    // ...
}
```

Se ejecuta la función *AsignarMem2D* y *x* toma un nuevo valor: la dirección del bloque de memoria reservado por **new**. ¿Ha cambiado *m* en el mismo valor? Evidentemente no, ya que tanto *x* como *m* son variables locales a sus respectivas funciones y no guardan ninguna relación una con la otra. Por lo tanto, *m* sigue valiendo 0, por lo que el programa no funciona. Además, cuando la función *AsignarMem2D* finalice, la variable local *x* será destruida y quedará un bloque de memoria sin referenciar y sin liberar (se ha generado una laguna de memoria).

Hagamos otra versión en la que se pase el puntero *m* por referencia, con la intención de que *AsignarMem2D* pueda acceder a su contenido y modificarlo almacenando en él la dirección del bloque de memoria reservado por ella.

```
// Punteros como parámetros
#include <iostream>
```

```
#include <iomanip>
using namespace std;

void AsignarMem2D(double ***, int, int);
void LiberarMem2D(double **, int);
void Visualizar(double **, int, int);

int main()
{
    int filas = 2, cols = 3; // número filas y columnas de la matriz
    double **m = 0;

    // Crear una matriz bidimensional dinámicamente
    AsignarMem2D(&m, filas, cols);
    if (m == 0)
    {
        cout << "Insuficiente memoria\n";
        return -1;
    }
    // Operaciones con la matriz
    // ...
    // Visualizar la matriz
    Visualizar(m, filas, cols);

    // Liberar la memoria asignada a la matriz
    LiberarMem2D(m, filas);
    return 0;
}

void AsignarMem2D(double ***x, int filas, int cols)
{
    double **p = 0; // dirección del bloque de memoria a reservar
    int f = 0;

    // Crear una matriz bidimensional dinámicamente
    // Matriz de punteros a cada una de las filas
    p = new (nothrow) double *[filas];
    if ( p == 0 ) { *x = 0; return; }
    // Iniciar la matriz de punteros con ceros
    fill(p, p+filas, static_cast<double *>(0));
    // Asignar memoria a cada fila
    for (f = 0; f < filas; f++)
    {
        p[f] = new (nothrow) double[cols];
        if ( p[f] == 0 )
        {
            LiberarMem2D(p, filas);
            *x = 0;
            return;
        }
    }
}
```

```

        // Iniciar la fila con ceros
        fill(p[f], p[f]+cols, 0);
    }
    *x = p; // guardar la dirección de la matriz en el parámetro
           // pasado por referencia a esta función.
           // El tipo de *x es double **, igual que el de p.
}

void LiberarMem2D(double **x, int filas)
{
    // Liberar la memoria asignada a la matriz
    for (int f = 0; f < filas; f++)
        delete [] x[f];
    delete [] x;
}

void Visualizar(double **x, int filas, int cols)
{
    for (int f = 0; f < filas; f++)
    {
        for (int c = 0; c < cols; c++)
            cout << setw(5) << x[f][c];
        cout << endl;
    }
}

```

Ahora, cuando **main** invoca a la función *AsignarMem2D*, ¿cómo se pasa el argumento *m*? Evidentemente por referencia, porque cuando se ejecuta la llamada se realiza la operación $x = \&m$. Por lo tanto, *AsignarMem2D* puede ahora acceder a *m*, puesto que conoce su dirección. Esto es, cuando *AsignarMem2D* quiera acceder a *m* lo hará a través de la expresión $*x$ (el contenido de la dirección x es m , puesto que x es la dirección de m).

Se puede observar que *AsignarMem2D* define un puntero p del mismo tipo que $*x$ y, por lo tanto, del mismo tipo que m , para guardar la dirección del bloque de memoria que hay que reservar para la matriz. Esto evitará tener que utilizar reiteradamente $*x$ en el resto de la función, lo que hará más fácil la interpretación del código. Por lo tanto, la última sentencia de esta función tiene que ser $*x = p$, que lo que hace en realidad es guardar p en m .

A diferencia de la función *AsignarMem2D*, las funciones *LiberarMem2D* y *Visualizar* son invocadas pasando m por valor, lo que es totalmente correcto porque ninguna de esas funciones necesita modificar la dirección m . Ahora si cualquiera de esas funciones ejecutara, por ejemplo, una operación como $x[0][0] = -1$, ¿el elemento $m[0][0]$ sería modificado en el mismo sentido? Evidentemente sí, porque x y m apuntan a la misma matriz.

Conclusión: cuando a una función se le pasa un puntero por valor, lógicamente los datos apuntados le son pasados por referencia, porque es la dirección de esos datos la que se le ha pasado, y cuando se le pasa un puntero por referencia, los datos apuntados también le son pasados por referencia.

Utilizando referencias, la función *AsignarMem2D* también podría escribirse así:

```
void AsignarMem2D(double **& x, int filas, int cols)
{
    int f = 0;

    // Crear una matriz bidimensional dinámicamente
    // Matriz de punteros a cada una de las filas
    x = new (nothrow) double *[filas];
    if ( x == 0 ) return;
    // Iniciar la matriz de punteros con ceros
    fill(x, x+filas, static_cast<double *>(0));
    // Asignar memoria a cada fila
    for (f = 0; f < filas; f++)
    {
        x[f] = new (nothrow) double[cols];
        if ( x[f] == 0 )
        {
            LiberarMem2D(x, filas);
            x = 0;
            return;
        }
        // Iniciar la fila con ceros
        fill(x[f], x[f]+cols, 0);
    }
}
```

En esta versión, el primer parámetro es una referencia (un sinónimo) al puntero a puntero que apunta a la matriz pasada como argumento.

PASAR UNA ESTRUCTURA A UNA FUNCIÓN

Una estructura puede ser pasada a una función, igual que cualquier otra variable, por valor o por referencia (en el capítulo anterior ya expusimos algo en relación con este tema). Cuando pasamos una estructura por valor, el parámetro actual que representa la estructura se copia en el correspondiente parámetro formal, produciéndose un duplicado de la estructura. Por eso, si alguno de los miembros del parámetro formal se modifica, estos cambios no afectan al parámetro actual correspondiente. Si pasamos la estructura por referencia, lo que recibe la función es

el lugar de la memoria donde se localiza dicha estructura. Entonces, conociendo su dirección, sí es factible alterar su contenido.

Como ejemplo vamos a realizar otra versión del programa anterior que utilice una estructura que defina la matriz; esto es, la estructura tendrá tres miembros: uno para almacenar la dirección de comienzo de la matriz, otro para almacenar el número de filas y otro más para el número de columnas.

```
struct tmatriz2D
{
    double **p; // dirección de comienzo de la matriz
    int filas; // número de filas
    int cols; // número de columnas
};
```

Supongamos que el planteamiento que hacemos para esta versión del programa, de forma esquemática, es así:

```
int main()
{
    tmatriz2D m = {0, 2, 3}; // estructura m

    AsignarMem2D(m);
    Visualizar(m);
    LiberarMem2D(m);
}

void AsignarMem2D(tmatriz2D x)
{
    x.p = new (nothrow) double *[x.filas];
    // ...
}

void LiberarMem2D(tmatriz2D x)
{
    // ...
}

void Visualizar(tmatriz2D x)
{
    // ...
}
```

Analicemos esta versión del programa. Cuando **main** invoca a la función *AsignarMem2D*, ¿cómo se pasa el argumento *m*? Es obvio que por valor, porque cuando se ejecuta la llamada se realiza la operación $x = m$. Esto hace que la estructura *m* se copie miembro a miembro en la estructura *x* del mismo tipo. Como *m.p* vale 0, *x.p* inicialmente también valdrá 0.

Se ejecuta la función *AsignarMem2D* y *x.p* toma un nuevo valor: la dirección del bloque de memoria reservado por **new**. ¿Ha cambiado *m.p* en el mismo valor? Evidentemente no, ya que *x* es una copia de *m*; ambas son estructuras locales a sus respectivas funciones y no guardan ninguna relación una con la otra. Por lo tanto, *m.p* sigue valiendo 0, por lo que el programa no funciona. Además, cuando la función *AsignarMem2D* finalice, la estructura *x* será destruida y quedará un bloque de memoria sin referenciar y sin liberar, generándose una laguna de memoria. ¿Cómo solucionamos este problema? Pues haciendo que *AsignarMem2D* pueda acceder al contenido del miembro *p* de *m* para modificarlo con la dirección del bloque de memoria reservado por ella. Para ello, necesita conocer el lugar en la memoria donde está ubicada la estructura *m*, esto es, su dirección, lo que implica pasar *m* por referencia.

```
// Estructuras como parámetros
// ...
void AsignarMem2D(tmatrix2D& x) // parámetro x pasado por referencia
{
    int f = 0, filas = x.filas, cols = x.cols;
    // Crear una matriz bidimensional dinámicamente
    // Matriz de punteros a cada una de las filas
    x.p = new (nothrow) double *[filas];
    if ( x.p == 0 ) return;
    // Iniciar la matriz de punteros con ceros
    fill(x.p, x.p+filas, static_cast<double *>(0));
    // Asignar memoria a cada fila
    for (f = 0; f < filas; f++)
    {
        x.p[f] = new (nothrow) double[cols];
        if ( x.p[f] == 0 )
        {
            LiberarMem2D(x);
            x.p = 0;
            return;
        }
        // Iniciar la fila con ceros
        fill(x.p[f], x.p[f]+cols, 0);
    }
}
```

Obsérvese que cuando **main** invoque a la función *AsignarMem2D* pasará el argumento *m* por referencia, porque el primer parámetro de esa función es una referencia (&) a una estructura *tmatrix2D*.

```
AsignarMem2D(m); // m es pasado por referencia
```

DATOS RETORNADOS POR UNA FUNCIÓN

Una función puede retornar cualquier valor de un tipo primitivo o derivado, excepto una matriz primitiva (estática) o una función. Cuando una función retorna un valor, lo que realmente devuelve es una copia de ese valor que, generalmente, se almacenará en una variable de su mismo tipo. Por lo tanto, se puede devolver un entero, una estructura, un objeto de cualquier clase, un puntero, una referencia, etc., pero no una matriz primitiva porque si éstas se pasan por referencia, también se devolverían por referencia; esto quiere decir que la función retornaría su dirección de comienzo y no una copia de sus elementos; después de lo expuesto, quizás esté pensando: si se devuelve la dirección de comienzo de la matriz, ¿no podríamos utilizar esa dirección para acceder a sus datos? Pues no, porque lógicamente, la matriz que tratamos de devolver habrá sido definida local a la función y, por lo tanto, será destruida cuando ésta finalice. Este mismo error se producirá siempre que devolvamos la dirección de cualquier variable local con la intención de acceder más tarde al valor que almacenaba.

Según lo expuesto, una función deberá devolver:

- Una copia de los datos o una referencia a los mismos.
- La dirección de un bloque de memoria reservado dinámicamente para contener los datos, o bien una referencia al mismo.
- La dirección de una variable **static**, o bien una referencia a la misma.

Retornar una copia de los datos

Como ejemplo vamos a realizar otra versión del programa anterior. En esta nueva versión sólo modificaremos, respecto a la versión anterior, la función *AsignarMem2D* para adaptarla al prototipo siguiente:

```
tmatriz2D AsignarMem2D(int, int);
```

Observamos que ahora *AsignarMem2D* tiene dos parámetros (el número de filas y de columnas de la matriz) y que devuelve una estructura de tipo *tmatriz2D*. Según esto, la función **main** y la función *AsignarMem2D* serán modificadas como se puede ver a continuación:

```
int main()
{
    int filas = 2, cols = 3;
    tmatriz2D m; // estructura m

    // Crear una matriz bidimensional dinámicamente
    m = AsignarMem2D(filas, cols);
```

```

    if (m.p == 0)
    {
        cout << "Insuficiente memoria\n";
        return -1;
    }

    // Operaciones con la matriz
    // ...

    // Visualizar la matriz
    Visualizar(m);

    // Liberar la memoria asignada a la matriz
    LiberarMem2D(m);
    return 0;
}

tmatriz2D AsignarMem2D(int filas, int cols)
{
    tmatriz2D x = {0, filas, cols};
    int f = 0;

    // Crear una matriz bidimensional dinámicamente
    // Matriz de punteros a cada una de las filas
    x.p = new (nothrow) double *[filas];
    if ( x.p == 0 ) return x;
    // Iniciar la matriz de punteros con ceros
    fill(x.p, x.p+filas, static_cast<double *>(0));
    // Asignar memoria a cada fila
    for (f = 0; f < x.filas; f++)
    {
        x.p[f] = new (nothrow) double[cols];
        if ( x.p[f] == 0 )
        {
            LiberarMem2D(x);
            x.p = 0;
            return x;
        }
        // Iniciar la fila con ceros
        fill(x.p[f], x.p[f]+cols, 0);
    }
    return x;
}

```

Se puede observar que ahora la función *AsignarMem2D* define una variable local *x* de tipo *tmatriz2D* y asigna a cada uno de sus miembros el valor correspondiente; esto es, a *p* el bloque de memoria reservado para la matriz (este bloque de memoria existirá mientras no sea liberado, operación que se puede realizar desde cualquier función que conozca su dirección), a *filas* el número de filas de la matriz

y a *cols* el número de columnas. Por otra parte, la función **main** invoca a *AsignarMem2D* para almacenar en *m* la estructura devuelta por dicha función. Como puede observar, no importa que *x* sea una variable local de *AsignarMem2D*, porque independientemente de que vaya a ser destruida, una vez finalizada la ejecución de la función, será copiada en *m*.

Retornar un puntero al bloque de datos

Continuando con el problema planteado en el apartado anterior, si estariamos cometiendo un error grave si hubiéramos escrito *AsignarMem2D* así:

```
int main()
{
    int filas = 2, cols = 3;
    tmatriz2D *m = 0; // puntero a una estructura tmatriz2D

    // Crear una matriz bidimensional dinámicamente
    m = AsignarMem2D(filas, cols);
    // ...
    return 0;
}

tmatriz2D *AsignarMem2D(int filas, int cols)
{
    tmatriz2D x = {0, filas, cols};
    int f = 0;

    // Matriz de punteros a cada una de las filas
    x.p = new (nothrow) double *[filas];
    // ...
    return &x;
}
```

Ahora, la función *AsignarMem2D* devuelve la dirección de la estructura *x*, que será almacenada por **main** en *m* una vez ejecutada esa función. Pero, ¿a quién apuntará *m* cuando *x* sea destruida una vez finalizada la ejecución de la función? La palabra destruida quiere decir que el espacio ocupado por los miembros de *x* es liberado, quedando dicho espacio disponible para el siguiente requerimiento de memoria que tenga el sistema; quiere esto decir que *m* almacenará una dirección que ya no es válida.

El error planteado puede ser subsanado si se reserva memoria dinámicamente para una variable de tipo *tmatriz2D* y se devuelve su dirección. No olvidar liberar ese bloque de memoria cuando la variable ya no sea necesaria. Con este nuevo planteamiento, la función **main** y la función *AsignarMem2D* quedarían así:

```
int main()
{
    int filas = 2, cols = 3;
    tmatriz2D *m = 0; // puntero a una estructura tmatriz2D
    // Crear una matriz bidimensional dinámicamente
    m = AsignarMem2D(filas, cols);
    if (m == 0)
    {
        cout << "Insuficiente memoria\n";
        return -1;
    }
    // Operaciones con la matriz
    // ...
    // Visualizar la matriz
    Visualizar(*m);
    // Liberar la memoria asignada a la matriz
    LiberarMem2D(*m);
    // Liberar la memoria asignada a la estructura
    delete m;
    return 0;
}

tmatriz2D *AsignarMem2D(int filas, int cols)
{
    int f = 0;
    tmatriz2D *x = new (nothrow) tmatriz2D;
    if ( x == 0 ) return 0;
    x->p = 0; x->filas = filas; x->cols = cols;

    // Crear una matriz bidimensional dinámicamente
    // Matriz de punteros a cada una de las filas
    x->p = new (nothrow) double *[filas];
    if ( x->p == 0 ) return 0;
    // Iniciar la matriz de punteros con ceros
    fill(x->p, x->p+filas, static_cast<double *>(0));
    // Asignar memoria a cada fila
    for ( f = 0; f < x->filas; f++)
    {
        x->p[f] = new (nothrow) double[cols];
        if ( x->p[f] == 0 )
        {
            LiberarMem2D(*x);
            x->p = 0;
            return 0;
        }
        // Iniciar la fila con ceros
        fill(x->p[f], x->p[f]+cols, 0);
    }
    return x;
}
```

Ahora, la función *AsignarMem2D* devuelve un puntero *x* a una estructura *tmatriz2D* creada dinámicamente, puntero que será almacenado por **main** en *m* una vez ejecutada esa función. Pero, ¿a quién apuntará *m* cuando *x* sea destruida una vez finalizada la ejecución de la función? La respuesta es a la misma estructura *tmatriz2D* que apuntaba *x*. En este caso, destruir *x* significa liberar los cuatro bytes que ocupaba esta variable tipo puntero, no liberar el bloque de memoria apuntado; de esto se encargará la función **main** al final del programa invocando al operador **delete**.

El mismo error se comete si la función *AsignarMem2D* devolviera una referencia a la estructura *x*:

```
int main()
{
    int filas = 2, cols = 3;
    tmatriz2D m; // estructura tmatriz2D
    // Crear una matriz bidimensional dinámicamente
    m = AsignarMem2D(filas, cols);
    // ...
    return 0;
}
```

```
tmatriz2D& AsignarMem2D(int filas, int cols)
{
    tmatriz2D x = {0, filas, cols};
    int f = 0;
    // Matriz de punteros a cada una de las filas
    //x.p = new (nothrow) double *[filas];
    // ...
    return x;
}
```

Retornar la dirección de una variable declarada static

Un usuario que utilice la última versión válida de *AsignarMem2D* deducirá, simplemente por el nombre y por el prototipo de la función, que la utilidad de dicha función es reservar memoria dinámicamente para una matriz de *filas* filas por *cols* columnas, lo cual deja claro también que dicha memoria hay que liberarla cuando ya no se necesite la matriz. En cambio, sólo con ver el prototipo no puede saber si dicha función utiliza también una estructura dinámica de datos de tipo *tmatriz2D* (lógicamente, este tipo de información deberá aparecer en la documentación de la función). En cualquier caso, resulta evidente que esto puede ser una fuente de errores, en el sentido de que si no se libera esa memoria se producirán lagunas de memoria. Para superar este inconveniente, podemos hacer que la función devuelva la dirección de una variable de tipo *tmatriz2D* declarada **static**, ya que este tipo

de variables persisten durante toda la ejecución del programa. Este mecanismo es el mismo que utiliza la función **localtime** de la biblioteca de C (ver apéndices).

Aplicando la teoría expuesta en el párrafo anterior, la función **main** y la función *AsignarMem2D* quedarían así:

```
int main()
{
    int filas = 2, cols = 3;
    tmatriz2D *m = 0; // estructura m

    // Crear una matriz bidimensional dinámicamente
    m = AsignarMem2D(filas, cols);
    if (m == 0)
    {
        cout << "Insuficiente memoria\n";
        return -1;
    }
    // Operaciones con la matriz
    // ...
    // Visualizar la matriz
    Visualizar(*m);

    // Liberar la memoria asignada a la matriz
    LiberarMem2D(*m);
    return 0;
}

tmatriz2D *AsignarMem2D(int filas, int cols)
{
    static tmatriz2D x;
    int f = 0;
    x.p = 0; x.filas = filas; x.cols = cols;

    // Crear una matriz bidimensional dinámicamente
    // Matriz de punteros a cada una de las filas
    x.p = new (nothrow) double *[filas];
    if ( x.p == 0 ) return 0;
    // Iniciar la matriz de punteros con ceros
    fill(x.p, x.p+filas, static_cast<double *>(0));
    // Asignar memoria a cada fila
    for (f = 0; f < x.filas; f++)
    {
        x.p[f] = new (nothrow) double[cols];
        if ( x.p[f] == 0 )
        {
            LiberarMem2D(x);
            x.p = 0;
            return 0;
        }
    }
}
```

```

    // Iniciar la fila con ceros
    fill(x.p[f], x.p[f]+cols, 0);
}
return &x;
}

```

Ahora, la función *AsignarMem2D* devuelve la dirección de la estructura *x* de tipo *tmatriz2D* declara **static**, dirección que será almacenada por **main** en *m* una vez ejecutada esa función. En este caso, cuando finalice el programa, la función **main** no tiene que liberar nada más que la memoria asignada para la matriz.

Retornar una referencia

Una función puede declararse para que devuelva una referencia, y hay dos razones para hacer esto:

1. Que la función se pueda utilizar a la izquierda del operador de asignación como sinónimo del objeto devuelto, lo que permitirá modificarlo.
2. Que el objeto que va a ser devuelto sea grande; en este caso, la devolución de una referencia puede ser más eficiente.

El siguiente ejemplo, muestra cómo implementar una función que retorne una referencia, y cómo utilizar posteriormente dicha función. Lo más normal es que este tipo de funciones sean métodos de una clase (véase el capítulo *Clases*), o bien de una estructura (**struct**), ya que una estructura es un caso particular de una clase en la que sus miembros son públicos por omisión.

```

// Retornar una referencia
#include <iostream>
using namespace std;

// Estructura de datos punto
struct punto
{
    // Atributos
    int x; // coordenada x
    int y; // coordenada y

    // Métodos
    int &cx() // devuelve una referencia a x
    {
        return x;
    }

    int &cy() // devuelve una referencia a y
    {

```

```
        return y;
    }
};
// Fin de la estructura de datos punto

int main()
{
    punto origen;

    // Utilizar cx() y cy() como l-values
    origen.cx() = 60;
    origen.cy() = 80;

    // Utilizar cx() y cy() como r-values
    cout << "x = " << origen.cx()
         << ", y = " << origen.cy() << endl;
}
```

Ejecución del programa

x = 60, y = 80

Obsérvese que el valor retornado por la función *cx* es una referencia al miembro *x* de la estructura *punto*. El resultado es que *cx* actúa como un nombre alternativo para *x*. Esto significa que una llamada a una función que retorna una referencia puede aparecer a la izquierda o a la derecha de un operador de asignación. Un razonamiento idéntico haríamos para *cy*.

Por otra parte, si pasar por referencia objetos grandes a una función es más eficiente que pasar una copia, también será más eficiente devolver por referencia un objeto grande, porque cuando una función devuelve un objeto necesita crear uno temporal, lo que la permite finalizar. Pero, devolver una referencia sólo será posible si el objeto no es destruido al finalizar la función. Por ejemplo, la siguiente función devuelve las coordenadas de un punto que se ha trasladado desde *p* a una nueva posición, sin perder las coordenadas, *cx* y *cy*, de su última posición:

```
punto& mover(punto p, int cx, int cy)
{
    p.cx() += cx;
    p.cy() += cy;
    return p; // objeto automático
    // Error: no se puede devolver una referencia a un objeto auto
}
```

Obsérvese que la función *mover* devuelve una referencia al nuevo punto *p*. Ahora bien, *p* es un objeto, que por ser local, dejará de existir cuando finalice la función. Por lo tanto, es un error devolver una referencia al objeto; en este caso sólo cabe devolver el objeto.

ARGUMENTOS EN LA LÍNEA DE ÓRDENES

Muchas veces, cuando invocamos a un programa desde el sistema operativo, necesitamos escribir uno o más argumentos a continuación del nombre del programa, separados por un espacio en blanco. Por ejemplo, piense en la orden `ls -l` del sistema operativo UNIX o en la orden `dir /p` de la consola de Windows. Tanto `ls` como `dir` son programas; `-l` y `/p` son opciones o argumentos en la línea de órdenes que pasamos al programa para que tenga un comportamiento diferente al que tiene de forma predeterminada; es decir, cuando no se pasan argumentos.

De la misma forma, nosotros podemos construir programas que admitan argumentos a través de la línea de órdenes. ¿Qué función recibirá esos argumentos? Lógicamente la función **main**, ya que es por esta función por donde empieza a ejecutarse un programa C++. Quiere esto decir que la función **main** tiene que tener parámetros formales donde se almacenen los argumentos pasados, igual que ocurre con cualquier otra función. Así, el prototipo de la función **main** en general es de la forma siguiente:

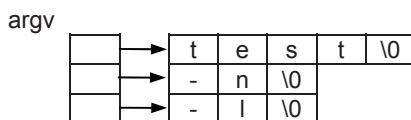
```
int main(int argc, char *argv[]);
```

El argumento `argc` es un entero que indica el número de argumentos pasados a través de la línea de órdenes, incluido el nombre del programa. El argumento `argv` es una matriz de punteros a cadenas de caracteres. Cada elemento de esta matriz apunta a un argumento, de manera que `argv[0]` contiene el nombre del programa, `argv[1]` el primer argumento de la línea de órdenes, `argv[2]` el segundo argumento, etc. La función **main** retorna un **int** con el que podemos expresar el éxito o no de la ejecución de dicha función.

Por ejemplo, supongamos que tenemos un programa C++ denominado `test` que acepta como argumentos `-n` y `-l`. Entonces, podríamos invocar a este programa escribiendo en la línea de órdenes del sistema operativo la siguiente orden:

```
test -n -l
```

Esto hace que `argc` tome automáticamente el valor 3 (nombre del programa más dos argumentos) y que el primer elemento de la matriz de punteros apunte al nombre del programa, y los dos siguientes a cada uno de los argumentos. Puede imaginar esta matriz de la forma siguiente:



Para clarificar lo expuesto vamos a realizar un programa que simplemente visualice los valores de los argumentos que se le han pasado en la línea de órdenes. Esto nos dará una idea de cómo acceder a esos argumentos. Supongamos que el programa se denomina *args* y que sólo admite los argumentos *-n*, *-k* y *-l*. Esto quiere decir que podremos especificar de cero a tres argumentos. Los argumentos repetidos y no válidos se desecharán. Por ejemplo, la siguiente línea invoca al programa *args* pasándole los argumentos *-n* y *-l*:

```
args -n -l
```

El código del programa propuesto se muestra a continuación.

```
// Argumentos en línea de órdenes
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    // Código común a todos los casos
    cout << "Argumentos:\n";
    if (argc == 1)
    {
        // Escriba aquí el código que sólo se debe ejecutar cuando
        // no se pasan argumentos
        cout << "    ninguno\n";
    }
    else
    {
        bool argumento_k = false, argumento_l = false, argumento_n = false;
        int i = 0;

        // ¿Qué argumentos se han pasado?
        for (i = 1; i < argc; i++)
        {
            if (strcmp(argv[i], "-k") == 0) argumento_k = true;
            if (strcmp(argv[i], "-l") == 0) argumento_l = true;
            if (strcmp(argv[i], "-n") == 0) argumento_n = true;
        }

        if (argumento_k) // si se pasó el argumento -k:
        {
            // Escriba aquí el código que sólo se debe ejecutar cuando
            // se pasa el argumento -k
            cout << "    -k\n";
        }

        if (argumento_l) // si se pasó el argumento -l:
        {
            // Escriba aquí el código que sólo se debe ejecutar cuando
```



```

    // se pasa el argumento -l
    cout << "    -l\n";
}

if (argumento_n) // si se pasó el argumento -n:
{
    // Escriba aquí el código que sólo se debe ejecutar cuando
    // se pasa el argumento -n
    cout << "    -n\n";
}
}
// Código común a todos los casos
}

```

Al ejecutar este programa, invocándolo como se ha indicado anteriormente, se obtendrá el siguiente resultado:

Argumentos:

```

-l
-n

```

La función **strcmp** de la biblioteca de C compara la *cadena1* con la *cadena2* lexicográficamente y devuelve un valor -1, 0 ó 1 dependiendo de que la primera cadena sea menor, igual o mayor, respectivamente, que la segunda.

REDIRECCIÓN DE LA ENTRADA Y DE LA SALIDA

Redireccionar la entrada significa que los datos pueden ser obtenidos de un medio diferente a la entrada estándar; por ejemplo, de un fichero en el disco. Si suponemos que tenemos un programa denominado *redir.cpp* que admite datos de la entrada estándar, la orden siguiente ejecutaría el programa *redir* y obtendría los datos de entrada de un fichero en el disco denominado *fdatos.ent*.

```
redir < fdatos.ent
```

Igualmente, redireccionar la salida significa enviar los resultados que produce un programa a un dispositivo diferente a la salida estándar; por ejemplo, a un fichero en disco. Tomando como ejemplo el programa *redir.cpp*, la orden siguiente ejecutaría el programa *redir* y escribiría los resultados en un fichero *fdatos.sal*.

```
redir > fdatos.sal
```

Observe que el programa se ejecuta desde la línea de órdenes, que para redireccionar la entrada se utiliza el símbolo “<” y que para redireccionar la salida se utiliza el “>”. También es posible redireccionar la entrada y la salida simultáneamente. Por ejemplo:

```
redir < fdatos.ent > fdatos.sal
```

Como aplicación de lo expuesto, vamos a realizar un programa que lea un conjunto de números y los escriba con un formato o con otro, en función de los argumentos pasados a través de la línea de órdenes. Esto es, si el programa se llama *redir.cpp*, la orden *redir* visualizará el conjunto de números sin más, pero la orden *redir -l* visualizará el conjunto de números escribiendo a continuación de cada uno de ellos un mensaje que indique si es par o impar. Por ejemplo:

```
24 es par
345 es impar
 7 es impar
... ..
```

El código de este programa se muestra a continuación.

```
// Redirección de la entrada-salida
#include <iostream>
#include <iomanip>
using namespace std;

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

    while (cin >> n)
    {
        cout << setw(6) << n;
        if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'l')
            cout << ((n%2) ? " es impar" : " es par");
        cout << endl;
    }
}
```

La solución que se obtiene al ejecutar este programa desde la línea de órdenes introduciendo los datos por el teclado es análoga a alguna de las dos siguientes:

```
redir[Entrar]
24 345 7 41 89 -72 5[Entrar]
 24
345
 7
 41
 89
-72
 5
[Ctrl]+Z]
```

```

redir -l[Entrar]
24 345 7 41 89 -72 5[Entrar]
  24 es par
  345 es impar
   7 es impar
  41 es impar
  89 es impar
 -72 es par
   5 es impar
[Ctrl]+Z

```

Observamos que si no se introduce el argumento *-l* simplemente se visualizan los valores tecleados y si se introduce, se visualizan los valores tecleados seguidos cada uno de ellos de la cadena “ es par” o “ es impar”, dependiendo de que el número sea par o impar.

También podemos editar un fichero *fdatos.ent* que contenga, por ejemplo, los datos:

```
24 345 7 41 89 -72 5
```

e invocar al programa *redir* de alguna de las formas siguientes:

```

redir [-l] < fdatos.ent
redir [-l] > fdatos.sal
redir [-l] < fdatos.ent > fdatos.sal

```

Los [] indican que opcionalmente se puede especificar el argumento *-l* (esta opción puede especificarse a continuación de *redir*, o bien al final de la línea). La primera orden leería los datos del fichero *fdatos.ent* y visualizaría los resultados por la pantalla, la segunda orden leería los datos del teclado y escribiría los resultados en el fichero *fdatos.sal* y la tercera orden leería los datos del fichero *fdatos.ent* y escribiría los resultados en el fichero *fdatos.sal*. Sepa que cuando se edita un fichero y se almacena, el sistema añade automáticamente, al final del mismo, la marca de fin de fichero.

FUNCIONES RECURSIVAS

Se dice que una función es recursiva si se llama a sí misma. El compilador C++ permite cualquier número de llamadas recursivas a una función. Cada vez que la función es llamada, los parámetros formales y las variables **auto** y **register** son iniciadas. Notar que las variables **static** solamente son iniciadas una vez, en la primera llamada.

¿Cuándo es eficaz escribir una función recursiva? La respuesta es sencilla: cuando el proceso a programar sea por definición recursivo. Por ejemplo, el cálculo del factorial de un número, $n! = n(n-1)!$, es por definición un proceso recursivo que se enuncia así: $factorial(n) = n * factorial(n-1)$.

Por lo tanto, la forma idónea de programar este problema es implementando una función recursiva. Como ejemplo, a continuación se muestra un programa que visualiza el factorial de un número. Para ello, se ha escrito una función *factorial* que recibe como parámetro un número entero positivo y devuelve como resultado el factorial de dicho número.

```
// Cálculo del factorial de un número
#include <iostream>
using namespace std;
unsigned long factorial(int n);

int main()
{
    int numero;
    unsigned long fac;
    do
    {
        cout << "¿Número? ";
        cin >> numero;
    }
    while (numero < 0 || numero > 25);
    fac = factorial(numero);
    cout << "\nEl factorial de " << numero << " es " << fac << endl;
}
```

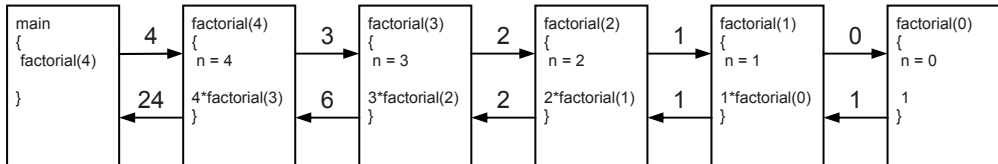
```
unsigned long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n*factorial(n-1);
}
```

En la tabla siguiente se ve el proceso seguido por la función *factorial*, durante su ejecución para $n = 4$.

Nivel de recursión Proceso de ida (pila de llamadas) Proceso de vuelta

0	factorial(4)		24
1	4 * factorial(3)	↓	4 * 6
2	3 * factorial(2)	↓	3 * 2
3	2 * factorial(1)	↓	2 * 1
4	1 * factorial(0)	↓	1 * 1

Cada llamada a la función factorial aumenta en una unidad el nivel de recursión. Cuando se llega a $n = 0$, se obtiene como resultado el valor 1 y se inicia la vuelta hacia el punto de partida, reduciendo el nivel de recursión en una unidad cada vez. La columna del centro especifica cómo crece la pila de llamadas hasta obtener un resultado que permita iniciar el retorno por la misma, dando solución a cada una de las llamadas pendientes. Gráficamente podemos representar este proceso así:



Se puede observar que la ejecución de *factorial* se inicia cinco veces; cuando se resuelve *factorial(0)* hay todavía cuatro llamadas pendientes de resolver; cuando se resuelve *factorial(1)* hay todavía tres llamadas pendientes de resolver; etc. Obsérvese también que el parámetro n es una variable local a la función, por eso está presente con su valor local en cada una de las ejecuciones.

Conclusión: por cada ejecución recursiva de la función, se necesita cierta cantidad de memoria para almacenar las variables locales y el estado en curso del proceso de cálculo con el fin de recuperar dichos datos cuando se acabe una ejecución y haya que reanudar la anterior. Por este motivo, en aplicaciones prácticas es imperativo demostrar que el nivel máximo de recursión es, no sólo finito, sino realmente pequeño.

Según lo expuesto, los algoritmos recursivos son particularmente apropiados cuando el problema a resolver o los datos a tratar se definen en forma recursiva. Sin embargo, el uso de la recursión debe evitarse cuando haya una solución obvia por iteración.

PARÁMETROS POR OMISIÓN EN UNA FUNCIÓN

Todos los parámetros formales de una función, o bien algunos de ellos, esto es, desde un determinado parámetro hasta el final, se pueden declarar por omisión. Es decir, en la declaración de la función o en su definición se especificarán los valores que deberán asumir los parámetros cuando se produzca una llamada a la función y éstos se omitan. Por ejemplo, la función *visualizar* que se expone a continuación asume para sus parámetros a , b y c los valores 1, 2.5 y 3.456, respectivamente, cuando éstos se omitan en la llamada.

```
// Parámetros por omisión
#include <iostream>
```

```

using namespace std;

void visualizar( int a = 1, float b = 2.5F, double c = 3.456 )
{
    cout << "parámetro 1 = " << a << ", "
         << "parámetro 2 = " << b << ", "
         << "parámetro 3 = " << c << endl;
}

int main()
{
    visualizar();
    visualizar( 2 );
    visualizar( 2, 3.7F );
    visualizar( 2, 3.7F, 8.125 );
}

```

Cuando ejecute este programa, obtendrá los siguientes resultados:

```

parámetro 1 = 1, parámetro 2 = 2.5, parámetro 3 = 3.456
parámetro 1 = 2, parámetro 2 = 2.5, parámetro 3 = 3.456
parámetro 1 = 2, parámetro 2 = 3.7, parámetro 3 = 3.456
parámetro 1 = 2, parámetro 2 = 3.7, parámetro 3 = 8.125

```

Observe que omitir un argumento en la llamada implica omitir todos los argumentos que le siguen y especificar los que le preceden.

Cuando utilicemos una declaración de función o función prototipo, la iniciación de los parámetros con los valores que deben asumir cuando estos se omitan en la llamada hay que realizarla sobre dicha función prototipo. Por ejemplo, la función *raíz* del siguiente programa devuelve la raíz enésima, dato que será aportado como segundo parámetro; por omisión es 2.

```

// Parámetros por omisión
#include <iostream>
#include <cmath>
using namespace std;

double raíz(double n, int = 2);

int main()
{
    cout << raíz(10) << endl; // raíz cuadrada por omisión
    cout << raíz(125, 3) << endl;
}

double raíz(double n, int r )
{
    if (n < 0) return 0; // "error: radicando negativo

```

```

    if (r < 1) return 0; // "error: raíz no válida
    return pow(n, 1.0/r);
}

```

La función **pow**(x, y) de la biblioteca de C da como resultado x^y . Cuando ejecute este programa, obtendrá los siguientes resultados:

```

3.16228
5

```

Observe en este ejemplo que el primer parámetro de la función *raíz* no tiene asignado un valor por defecto, por lo que en la llamada habrá siempre que especificar al menos ese valor.

FUNCIONES EN LÍNEA

Cuando una *función* se califica *en línea* (**inline**) el compilador tiene la facultad de reemplazar cualquier llamada a la función en el programa fuente por el cuerpo actual de la función. Quiere esto decir que el compilador puede tomar la iniciativa de no expandir la función; por ejemplo, por ser demasiado larga.

Para poder asignar el calificativo de *en línea* a una *función*, dicha función debe estar definida antes de que sea invocada, de lo contrario el compilador no lo tendrá en cuenta. Esta es la razón por la que las funciones **inline** son normalmente definidas en ficheros de cabecera.

Calificar a una función *en línea* implica anteponer el calificativo **inline** al tipo retornado por la función. Por ejemplo:

```

inline int menor( int x, int y )
{
    return x < y ? x : y ;
}

```

Con las funciones **inline** se obtienen tiempos de ejecución más bajos, ya que se evitan las llamadas a cada una de estas funciones. No obstante, el abuso de este calificador en ocasiones puede no ser bueno. Por ejemplo, la modificación de una función **inline** obligaría a recompilar todos los módulos en los que ésta apareciera. Por otra parte, el tamaño del código puede aumentar extraordinariamente. Por todo ello, se recomienda utilizar el calificador **inline** cuando la función es muy pequeña, o si se llama desde pocos lugares.

Cualquier función definida en la declaración de una clase se asume como una función **inline**.

MACROS

El comportamiento de las macros declaradas con la directriz **#define** es similar al de las funciones **inline**. Sin embargo, es mejor utilizar funciones **inline** que macros, ya que sus parámetros son chequeados automáticamente y no presentan los problemas de las macros parametrizadas. Por ejemplo:

```
// Una macro comparada con una función en línea
#include <iostream>
using namespace std;

#define MENOR( X, Y ) ((X) < (Y) ? (X) : (Y))

inline int menor( int x, int y )
{
    return x < y ? x : y;
}

int main()
{
    int m, a = 10, b = 20;

    m = MENOR( a--, b-- ); // efecto colateral
                          // el valor menor se decrementa dos veces
    cout << "menor = " << m << ", a = " << a << ", b = " << b << endl;

    a = 10; b = 20;
    m = menor( a--, b-- );
    cout << "menor = " << m << ", a = " << a << ", b = " << b << endl;
}
```

Este ejemplo da lugar al siguiente resultado:

```
menor = 9, a = 8, b = 19
menor = 10, a = 9, b = 19
```

Después de la sustitución de la macro, la sentencia resultante es así:

```
m = ((a--) < (b--)) ? (a--) : (b--);
```

La ejecución de esta sentencia se desarrolla de la forma siguiente:

$$1. \quad ((a--) < (b--)) \begin{cases} a < b \\ a-- \\ b-- \end{cases}$$

$$2. \text{ Si fue } a < b, m = a \text{ -- } \begin{cases} m = a \\ a \text{ --} \end{cases}$$

$$3. \text{ Si no fue } a < b, m = b \text{ -- } \begin{cases} m = b \\ b \text{ --} \end{cases}$$

Aplicando lo expuesto a nuestro ejemplo, se compara 10 y 20; el resultado es menor; se decrementa a a 9 y b a 19; como a fue menor que b , se asigna a a m , valor 9, y se decrementa a a 8. El resultado es $m = 9$, $a = 8$ y $b = 19$.

Esto indica que las macros son muy importantes en C pero no en C++ en donde pueden ser sustituidas por funciones **inline**.

FUNCIONES SOBRECARGADAS

Normalmente, cada función tiene su propio nombre que la distingue de las demás. No obstante, se pueden presentar casos en los que varias funciones ejecuten la misma tarea sobre objetos de diferentes tipos, y puede resultar conveniente que dichas funciones tengan el mismo nombre. En este caso, se dice que la función está sobrecargada.

La sobrecarga de una función es una característica de C++ que hace los programas más legibles. Consiste en volver a declarar una función ya declarada, con distinto número y/o tipo de parámetros. Una función sobrecargada no puede diferir solamente en el tipo del resultado, sino que debe diferir también en el tipo y/o en el número de sus parámetros formales.

La sobrecarga de una función sólo puede ocurrir dentro de su mismo ámbito.

Por ejemplo, supongamos una función *visualizar* para mostrar expresiones al estilo de **cout** pero de una forma más sencilla. Podemos diseñar tantas funciones como casos pensemos que un usuario pueda necesitar. Por ejemplo, escribamos el siguiente fichero de cabecera:

```
// MisFunciones - Fichero de cabecera
#include <string>
using namespace std;

void visualizar(string cad = "\n");
void visualizar(long n, char car = '\n');
void visualizar(string cad, long n, char car = '\n');
void visualizar(double n, char car = '\n');
void visualizar(string cad, double n, char car = '\n');
```

Cuando la función *visualizar* es llamada, el compilador debe resolver cuál de las funciones con el nombre *visualizar* es invocada. Esto lo hace comparando los tipos de los parámetros actuales con los tipos de los parámetros formales de todas las funciones llamadas *visualizar*. Si no encontrara una función exactamente con los mismos tipos de argumentos, realizaría las conversiones permitidas sobre los parámetros actuales, buscando así una función apropiada.

Escribamos ahora un fichero *MisFunciones.cpp* con las definiciones de las funciones *visualizar* descritas anteriormente. Esto es:

```
// MisFunciones.cpp - Definiciones de MisFunciones
#include <iostream>
#include <string>
using namespace std;

void visualizar(string cad)
{
    cout << cad;
}
void visualizar(long n, char car)
{
    cout << n << car;
}
void visualizar(string cad, long n, char car)
{
    cout << cad << n << car;
}
void visualizar(double n, char car)
{
    cout << n << car;
}
void visualizar(string cad, double n, char car)
{
    cout << cad << n << car;
}
```

Ahora, utilizando el fichero de cabecera *MisFunciones*, vamos a escribir un programa que permita escribir distintos resultados. La idea es que el programa invoque automáticamente a una u otra función, dependiendo de los argumentos pasados en la llamada.

```
// Funciones sobrecargadas
#include "MisFunciones"

int main()
{
    long ai = 2, bi = 2;
    double ad = 1.5;
```

```

visualizar("Resultados: ");           // invoca a la primera función
visualizar(ai);                       // invoca a la segunda función
visualizar("Dato entero = ", bi);     // invoca a la tercera función
visualizar("Dato real = ", ad);      // invoca a la quinta función
}

```

Para compilar el programa ejecute desde la línea de órdenes una orden análoga a la siguiente, o bien realice un proyecto equivalente:

```
g++ main.cpp misfunciones.cpp -o fnsobrecargada.exe
```

Ambigüedades

Supongamos que ahora hemos escrito este otro programa:

```

#include "MisFunciones"
int main()
{
    int ai = 2;
    float ad = 3.4F;
    visualizar(ai);
    visualizar("Dato real = ", ad);
}

```

Aunque los prototipos de *visualizar* son diferentes, las llamadas realizadas en este programa a dicha función presentan diferentes comportamientos:

- La llamada *visualizar(ai)* es ambigua y produce un error durante compilación, porque, al no existir ninguna sobrecarga con un primer parámetro de tipo **int**, son candidatas a ser ejecutadas las siguientes funciones:

```

void visualizar(long int, char = '\n')
void visualizar(double, char = '\n')

```

ya que un **int** puede también ser convertido implícitamente a un **long**, y a un **double**.

- La llamada *visualizar("Dato real = ", ad)* no es ambigua porque sólo es candidata a ser ejecutada la función:

```
void visualizar(string, double, char = '\n')
```

La solución pasa por hacer una conversión explícita según sea nuestra intención. Por ejemplo:

```
visualizar(static_cast<long>(ai));
```

OPERADORES SOBRECARGADOS

Suele ser útil asociar funciones con operaciones para habilitar el uso de la notación convencional del operador que define esa operación. Se trata de un caso particular de funciones sobrecargadas. En estos casos el nombre de la función debe estar formado por la palabra reservada **operator** más el operador. Por ejemplo, el siguiente programa sobrecarga el operador `+` para permitir sumar estructuras de datos que representan complejos.

```
// Operadores sobrecargados
#include <iostream>
using namespace std;

// Estructura de un número complejo
struct complejo
{
    double real, imag;
};

// Suma de números complejos
complejo operator+( const complejo x, const complejo y )
{
    complejo temp;
    temp.real = x.real + y.real; // parte real
    temp.imag = x.imag + y.imag; // parte imaginaria
    return temp;
}

// Visualizar un número complejo
void visualizar( const complejo &z )
{
    cout << '(' << z.real << ',' << z.imag << ')' << endl;
}

int main()
{
    complejo a = { 1.0, 2.0 }, b = { 1.5, -1.5 }, c;

    // Sumar los complejos a y b. Forma abreviada.
    c = a + b;
    visualizar(c);

    // Sumar los complejos a y b. Llamada explícita.
    c = operator+(a, b);
    visualizar(c);
}
```

Para saber cómo trabajan los operadores sobrecargados, observe en el ejemplo anterior la sentencia $c = a + b$. Ahora el operador $+$, además de tener la funcionalidad que todos conocemos, esto es, sumar números enteros, fraccionarios, etc., tiene una funcionalidad añadida, sumar números complejos, provista por la función **operator+**. Cuando ambos operandos, a y b , sean de tipo *complejo*, $+$ invocará automáticamente a la función **operator+** para sumarlos. Esto nos permite sumar números complejos utilizando la misma notación que la utilizada para sumar números enteros, por ejemplo.

Obsérvese también que la expresión $a + b$ es equivalente a la llamada explícita: `operator+(a, b)`.

Un operador sobrecargado no puede tener argumentos por omisión. Para implementar otros operadores, siga el mismo procedimiento.

PUNTEROS A FUNCIONES

Igual que sucedía con las matrices primitivas, el nombre de una función representa la dirección donde se localiza esa función; quiere esto decir que como tal dirección, puede pasarse como argumento a una función, almacenarla en un puntero (que puede ser un elemento de una matriz), etc. La sintaxis para declarar un puntero a una función es así:

```
tipo (*p_identif)(parms);
```

donde *tipo* es el tipo del valor devuelto por la función, *parms* son las declaraciones de los parámetros de la función y *p_identif* es el nombre de una variable de tipo puntero. Esta variable almacenará la dirección de comienzo de una función, dada por el propio nombre de la función. Por ejemplo:

```
double (*pfn)();
```

indica que *pfn* es un puntero a una función que devuelve un valor de tipo **double**. Observe en la declaración los paréntesis que envuelven al identificador; son fundamentales. Si no los ponemos, el significado cambia completamente. Veamos:

```
double *pfn();
```

indica que *pfn* es una función que devuelve un puntero a un valor de tipo **double**.

Siguiendo con la declaración de *pfn*, sabemos que la función apuntada tiene que devolver un valor de tipo **double**, pero, ¿qué argumentos tiene? La respuesta es que no tiene argumentos porque en C++ los paréntesis vacíos o paréntesis con **void** significan lo mismo: función sin argumentos (en C la ausencia de argumen-

tos indica cualquier número y tipo de argumentos, en cambio si entre los paréntesis escribimos **void**, significa sin argumentos). Por lo tanto, la función que se asigne a *pfn* tiene que tener el mismo prototipo que la siguiente:

```
double f1();
```

Esta función puede ser asignada a *pfn* como se muestra a continuación:

```
pfn = &f1; // correcto
pfn = f1; // correcto; la utilización de & es opcional.
```

Finalmente, para invocar a la función apuntada por *pfn*, puede utilizarse cualquiera de las dos formas siguientes:

```
(*pfn)(); // correcto
pfn();    // correcto; la utilización de * es opcional.
```

Resumiendo: en las asignaciones de punteros a funciones, el tipo de la función debe coincidir exactamente con el tipo del puntero. Un ejemplo:

```
void fa(string cadena);
void fb(string cadena);
void fc(int n);
int fd(string cadena);
void (*pfn)(string); // pfn: puntero a función

int main()
{
    pfn = fa; // correcto
    pfn = fb; // correcto
    pfn = fc; // error: tipo de parámetro erróneo
    pfn = fd; // error: tipo de retorno erróneo
}
```

También, suele ser conveniente definir un nombre para el tipo de puntero a función para evitar la utilización de la sintaxis de puntero a función. Por ejemplo:

```
void error(string cadena); // función error
void opcion(string cadena); // función opción
typedef void (*pfn)(string); // tipo pfn

int main()
{
    pfn pfn1 = error; // pfn1 apunta a la función error
    pfn pfn2 = opcion; // pfn2 apunta a la función opción
    // ...
}
```

Así mismo, en la mayoría de las ocasiones suele ser más útil una matriz de punteros a funciones. Por ejemplo:

```
void error(string cadena); // función error
void opcion(string cadena); // función opción
typedef void (*pfn)(string); // tipo pfn

int main()
{
    pfn mpfn[] = { error, opcion };
    // ...
    mpfn[0]("error: valor negativo"); // invoca a la función error
    mpfn[1]("leer"); // invoca a la función opción
    // ...
}
```

El programa siguiente muestra un ejemplo de utilización de punteros a funciones. Este programa permite buscar y escribir el valor menor de una lista de datos numérica o de una lista de datos alfanumérica. Si en la línea de órdenes se especifica un argumento “-n”, se interpreta la lista como numérica; en los demás casos se interpreta como alfanumérica. Por ejemplo, si el programa se llama *PtrFn* la orden:

```
PtrFn -n
```

invoca al programa para trabajar con una lista de datos numérica.

El programa consta básicamente de una función *fmenor* que busca en una lista de datos el menor, independientemente del tipo de comparación (numérica o alfanumérica). Para ello, la función *fmenor* recibe como parámetro una función, la adecuada para comparar los datos según sean estos numéricos o alfanuméricos. Para comparar datos numéricos, se utiliza la función *compnu* y para comparar datos alfanuméricos, se utiliza la función *compal*.

```
string fmenor(vector<string> cadena, pfn comparar)
{
    // Buscar el dato menor de una lista
    string menor;

    int i = 0, c = cadena.size();
    menor = cadena[i]; // menor = primer dato
    while ( --c > 0)
        // comparar menor con el siguiente dato
        menor = comparar(menor, cadena[++i]);

    return menor;
}
```

El argumento *cadena* es la matriz de datos (cadenas de caracteres numéricas o alfanuméricas) y *comparar* es un puntero a una función que hace referencia a la función utilizada para comparar (*compnu* o *compal*). La llamada a la función *fmenor* es de la forma siguiente:

```
if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
    dato = fmenor(cadena, compnu);
else
    dato = fmenor(cadena, compal);
```

El programa completo se muestra a continuación.

```
// Punteros a funciones
#include <iostream>
#include <string>
#include <vector>
using namespace std;

string compnu(string x, string y);
string compal(string x, string y);

typedef string (*pfn)(string, string);
string fmenor(vector<string>, pfn);

int main(int argc, char *argv[])
{
    vector<string> cadena; // matriz de punteros a los datos
    cadena.reserve(100); // reservar un espacio inicial
    string dato; // dato
    int c = 0; // contador

    // Leer la lista de datos numéricos o alfanuméricos
    cout << "Introducir datos y finalizar con eof\n\n";
    cout << "Dato " << ++c << ": ";
    getline(cin, dato);
    while (!cin.fail())
    {
        cadena.push_back(dato);
        cout << "Dato " << ++c << ": ";
        getline(cin, dato);
    }
    cin.clear();

    // argv[1] != "-n" -> búsqueda en una lista alfanumérica,
    // argv[1] = "-n" -> búsqueda en una lista numérica

    if (argc > 1 && argv[1][0] == '-' && argv[1][1] == 'n')
    {
        cout << "Tipo de comparación: numérica.\n";
    }
}
```



```

    dato = fmenor(cadena, compnu);
}
else
{
    cout << "Tipo de comparación: alfanumérica.\n";
    dato = fmenor(cadena, compal);
}
cout << "\nEl elemento menor de la lista es: " << dato << endl;
}

```

```

string fmenor(vector<string> cadena, pfn comparar)
{
    // Buscar el dato menor de una lista
    string menor;

    int i = 0, c = cadena.size();
    menor = cadena[i]; // menor = primer dato
    while ( --c > 0)
        // comparar menor con el siguiente dato
        menor = comparar(menor, cadena[++i]);

    return menor;
}

```

```

string compnu(string x, string y)
{
    // Comparar dos datos numéricamente
    if (atof(x.c_str()) > atof(y.c_str()))
        return y;
    else
        return x;
}

```

```

string compal(string x, string y)
{
    // Comparar dos datos alfanuméricamente
    if (x < y)
        return x;
    else
        return y;
}

```

Ejecución del programa:

```

puntsfns -n
Introducir datos y finalizar con eof

```

```

Dato 1: 123
Dato 2: -7.5
Dato 3: 23

```

```
Dato 4: -1.35
Dato 5: 45
Dato 6: ^Z
Tipo de comparación: numérica.
```

El elemento menor de la lista es: -7.5

EJERCICIOS RESUELTOS

1. Escribir una función que, partiendo de dos matrices de cadenas de caracteres ordenadas en orden ascendente, construya una tercera matriz también ordenada en orden ascendente. La idea que se persigue es construir la tercera lista ordenada; no construirla y después ordenarla mediante una función.

Para ello, la función **main** proporcionará las dos matrices e invocará a una función cuyo prototipo será el siguiente:

```
int Fusionar(const vector<string>&, const vector<string>&, vector<string>&);
```

El primer parámetro de la función *Fusionar* y el segundo son las matrices de partida; el parámetro tercero es la matriz que almacenará los elementos ordenados de las dos anteriores.

El proceso de fusión consiste en:

- a) Partiendo de que ya están construidas las dos matrices de partida, tomar un elemento de cada una de las matrices.
- b) Comparar los dos elementos (uno de cada matriz) y almacenar en la matriz resultado el menor.
- c) Tomar el siguiente elemento de la matriz a la que pertenecía el elemento almacenado en la matriz resultado, y volver al punto b).
- d) Cuando no queden más elementos en una de las dos matrices de partida, se copian directamente en la matriz resultado todos los elementos que queden en la otra matriz.

Las funciones *Fusionar* y **main** se muestran a continuación, el resto ya han sido realizadas en capítulos anteriores.

```
// Fusionar dos listas clasificadas
#include <iostream>
#include <string>
#include <vector>
```

```
using namespace std;

int Fusionar(const vector<string>&, const vector<string>&,
            vector<string>&);
void Leer(vector<string>&);
void Ordenar(vector<string>&);
void Mostrar(vector<string>&);
void Error(void);

int main()
{
    // Crear las listas 1 y 2, leer los datos, y ordenarlas
    vector<string> lista1;
    Leer(lista1);
    Ordenar(lista1);

    vector<string> lista2;
    Leer(lista2);
    Ordenar(lista2);

    // Declarar la lista resultante de fusionar las anteriores
    vector<string> lista3;

    // Fusionar lista1 y lista2 y almacenar el resultado en
    // lista3. La función "fusionar" devuelve un 0 si no se
    // pudo realizar la fusión.

    int r = Fusionar(lista1, lista2, lista3);

    // Mostrar la matriz resultante
    if (r)
        Mostrar(lista3);
    else
        Error();
}

int Fusionar(const vector<string>& lista1,
            const vector<string>& lista2,
            vector<string>& lista3)
{
    unsigned int i = 0, i1 = 0, i2 = 0, i3 = 0;
    size_t tam1 = lista1.size();
    size_t tam2 = lista2.size();
    if (tam1 == 0 && tam2 == 0) return 0;

    while (i1 < tam1 && i2 < tam2)
        if (lista1[i1] < lista2[i2])
            lista3.push_back(lista1[i1++]);
        else
            lista3.push_back(lista2[i2++]);
}
```

```
// Los dos lazos siguientes son para prever el caso de que,
// lógicamente una lista finalizará antes que la otra.

for (i = i1; i < tam1; i++)
    lista3.push_back(lista1[i]);

for (i = i2; i < tam2; i++)
    lista3.push_back(lista2[i]);

return l;
}

void Leer(vector<string>& lista)
{
    cout << "Escriba las cadenas que desea introducir.\n";
    cout << "Puede finalizar con <eof>\n";
    int fila = 0;
    string cad; // una lista

    cout << "lista[" << fila++ << "]: ";
    getline(cin, cad);
    while (!cin.eof()) // si se pulsó [Ctrl][z], salir del bucle
    {
        lista.push_back(cad); // añadir una lista
        cout << "lista[" << fila++ << "]: ";
        getline(cin, cad); // leer otra lista
    }
    cin.clear(); // desactivar el indicador de eof
}

void Ordenar(vector<string>& lista)
{
    string aux;
    int i = 0, k = 0;
    size_t ncad = lista.size();

    // Ordenar: método de inserción
    for (i = 1; i < ncad; i++)
    {
        aux = lista[i];
        k = i - 1;
        while ((k >= 0) && aux < lista[k])
        {
            lista[k+1] = lista[k];
            k--;
        }
        lista[k+1] = aux;
    }
}
```

```

void Mostrar(vector<string>& lista)
{
    vector<string>::iterator cad; // cad es un iterador
    for (cad = lista.begin(); cad != lista.end(); ++cad)
        cout << *cad << '\n'; // mostrar una lista
}

void Error(void)
{
    cerr << "Error: las listas están vacías\n";
    exit(1);
}

```

2. El cálculo de los números de Fibonacci es un ejemplo de una definición matemática recursiva que se enuncia así: el número de Fibonacci $f(i)$, siendo i el número de orden (0, 1, 2, 3, 4, 5, ...) del número a calcular, es igual al número de Fibonacci $f(i-1)$ más el número de Fibonacci $f(i-2)$, sabiendo que $f(0)$ es 0 y $f(1)$ 1.

```

f(0) = 0
f(1) = 1
f(2) = f(1) + f(0)
f(3) = f(2) + f(1)
...
f(i) = f(i-1) + f(i-2)

```

Realizar un programa que pregunte: ¿cuántos números de Fibonacci, a partir del primero, se quieren calcular?, almacene esos números en una matriz del tamaño necesario y finalmente los muestre. Para ello se deberá utilizar una función recursiva con el prototipo indicado a continuación:

```
int fibonacci(int n);
```

La función *fibonacci* devolverá como resultado el número de Fibonacci cuyo número de orden (0, 1, 2, ...) sea n .

El programa completo se muestra a continuación.

```

// Secuencia de Fibonacci
#include <iostream>
#include <iomanip>
#include <limits>
#include <vector>
using namespace std;
int fibonacci(int);
unsigned int leerDato();

int main()
{

```

```
int n = 0, i = 0;

cout << "¿Cuántos números de Fibonacci, a partir del ";
cout << "primero, se quieren calcular?\n";

n = leerDato();

// Crear una matriz dinámicamente
vector<int> f(n);

// Obtener los números de la serie
for (i = 0; i < n; i++)
    f[i] = fibonacci(i);

// Visualizar la matriz
for (i = 0; i < n; i++)
    cout << setw(5) << f[i];
cout << endl;
}

int fibonacci(int n)
{
    if ( n == 0 )
        return 0;
    else if ( n == 1 )
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}

unsigned int leerDato()
{
    unsigned int dato = 0;
    cin >> dato;
    while (cin.fail()) // si el dato es incorrecto, limpiar el
    {
        // búfer y volverlo a leer
        cout << '\a'; // bip
        cin.clear(); // desactivar los indicadores de error
        cin.ignore(numeric_limits<int>::max(), '\n');
        cin >> dato;
    }
    // Eliminar posibles caracteres sobrantes
    cin.ignore(numeric_limits<int>::max(), '\n');
    return dato;
}
```

Ejecución del programa:

¿Cuántos números de Fibonacci, a partir del primero, se quieren calcular? 10
0 1 1 2 3 5 8 13 21 34

Éste es un ejemplo donde el uso de la recursión puede evitarse porque hay una solución obvia por iteración, que dará lugar a una ejecución más rápida y con un coste de recursos de memoria bastante inferior, ejercicio que se propone en el siguiente apartado.

EJERCICIOS PROPUESTOS

1. Responda a las siguientes preguntas:

1) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
using namespace std;
int[] test();

int main()
{
    int *p = 0, i = 0;
    p = test();
    for (i = 0; i < 3; i++)
        cout << p[i] << ' ';
}

int[] test()
{
    static int a[] = {1, 2, 3};
    return a;
}
```

- a) 0 0 0.
- b) 1 2 3.
- c) No se puede ejecutar porque hay errores durante la compilación.
- d) No se puede ejecutar porque hay errores durante la ejecución.

2) ¿Cuál es el resultado del siguiente programa?

```
#include <iostream>
using namespace std;
int *test();

int main()
{
    int *p = 0, i = 0;
    p = test();
    for (i = 0; i < sizeof(p)/sizeof(int); i++)
        cout << p[i] << ' ';
}
```

```
int *test()
{
    static int a[] = {1, 2, 3};
    return a;
}
```

- a) 1.
- b) 1 2 3.
- c) Imprime basura (valor no predecible).
- d) Produce un error durante la ejecución.

3) ¿Cuál es el resultado del siguiente programa?

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

void test(vector<int> a)
{
    for (unsigned int i = 0; i < a.size(); i++)
        a[i]++;
}

int main()
{
    vector<int> a(3);
    test(a);
    for (unsigned int i = 0; i < a.size(); i++)
        cout << a[i] << ' ';
}
```

- a) 1 1 1.
- b) 0 0 0.
- c) Imprime basura (valores no predecibles).
- d) Ninguno de los anteriores.

4) ¿Cuál es el resultado del siguiente programa?

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

void test(vector<int> x)
{
    for (unsigned int i = 0; i < x.size(); i++)
        cout << x[i] << ' ';
}
```



```
int main()
{
    vector<int> a;
    a.reserve(3);
    test(a);
}
```

- a) 0 0 0.
- b) Imprime basura (valores no predecibles).
- c) El programa produce un error durante la compilación.
- d) No imprime nada.

5) ¿Cuál es el resultado del siguiente programa?

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

struct tmatriz1D
{
    int *p;
    int n;
};

void test(tmatriz1D s)
{
    s.p = new (nothrow) int[s.n];
    if (s.p != 0)
        fill(s.p, s.p+s.n, 1);
}

int main()
{
    tmatriz1D s = {0, 3};
    test(s);
    for (int i = 0; i < s.n; i++)
        cout << s.p[i] << ' ';
    delete [] s.p;
}
```

- a) 0 0 0.
- b) 1 1 1.
- c) El programa produce un error durante la compilación.
- d) El programa produce un error durante la ejecución.

6) ¿Cuál es el resultado del siguiente programa?

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

```
vector<int>& test()
{
    vector<int> a(3);
    a[0] = 1; a[1] = 2; a[2] = 3;
    return a;
}

int main()
{
    vector<int>& v = test();
    for (unsigned int i = 0; i < v.size(); i++)
        cout << v[i] << ' ';
}
```

- a) 1 2 3.
 - b) Imprime basura (valores no predecibles).
 - c) 0 0 0.
 - d) La función *test* no debe devolver una referencia.
- 7) ¿Cuál es el valor de los parámetros *argc* y *argv* cuando se invoque a este programa mediante la orden *nombre_programa test.txt*?

```
int main(int argc, char *argv[])
{
    // ..
}
```

- a) *argc* = 1, *argv*[0] = nombre_programa.
 - b) *argc* = 1, *argv*[0] = text.txt.
 - c) *argc* = 2, *argv*[0] = nombre_programa, *argv*[1] = text.txt.
 - d) Ninguna de las anteriores.
- 8) ¿Cuál es el resultado del siguiente programa?

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

void test(int x)
{
    if (x) test(x-1);
    cout << x << ' ';
}

int main()
{
    test(5);
}
```

- a) 5 4 3 2 1 0.

- b) 0 1 2 3 4 5.
- c) 0 2 4.
- d) 1 3 5.

9) ¿Cuál es el resultado del siguiente programa?

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

void test(char *c1, char *c2)
{
    if (*c2)
    {
        *c1++ = *c2++;
        test(c1, c2);
    }
    else
        *c1 = 0;
}

int main()
{
    char cad1[80], cad2[] = "hola";
    test(cad1, cad2);
    cout << cad1 << '\n';
}
```

- a) hola.
- b) aloh.
- c) Bucle infinito.
- d) Produce un error durante la ejecución.

10) ¿Cuál es el resultado del siguiente programa?

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

void test(string c = "xxxx")
{
    int n = c.length();
    while (--n != -1)
        cout << c[n];
}

int main()
{
    string cad = "hola";
    test();
}
```

- a) aloh.
 - b) hola.
 - c) xxxx.
 - d) Produce un error durante la compilación.
2. Calcular números de Fibonacci. El número de Fibonacci $f(i)$, siendo i el número de orden (0, 1, 2, 3, ...), es igual al número de Fibonacci $f(i-1)$ más el número de Fibonacci $f(i-2)$, partiendo de que $f(0)$ vale 0 y $f(1)$ vale 1.

```
f(0) = 0
f(1) = 1
f(2) = f(1) + f(0)
f(3) = f(2) + f(1)
...
f(i) = f(i-1) + f(i-2)
```

Realizar un programa que pregunte: ¿cuántos números de Fibonacci, a partir del primero, se quieren calcular?, almacene esos números en una matriz del tamaño necesario y finalmente los muestre. Para ello se deberá utilizar una función NO RECURSIVA con el prototipo indicado a continuación:

```
int fibonacci(int n);
```

La función *fibonacci* devolverá como resultado el número de Fibonacci cuyo número de orden (0, 1, 2, ...) sea n .

3. Suponiendo un texto escrito en minúsculas y sin signos de puntuación, es decir, una palabra estará separada de otra por un espacio en blanco, realizar un programa que lea texto de la entrada estándar (del teclado) y dé como resultado la frecuencia con que aparece cada palabra leída del texto. El resultado se almacenará en una matriz en la que cada elemento será una estructura del tipo siguiente:

```
struct telem
{
    string palabra; // palabra
    int contador;  // número de veces que aparece en el texto
};
```

La estructura del programa estará formada por las funciones **main** y siguientes:

```
bool EstaPalabra(vector<telem>&, telem);
void InsertarPalabra(vector<telem>&, telem);
void VisualizarMatriz(vector<telem>&);
```

La función **main** creará una matriz inicialmente con cero elementos, y utilizando las funciones anteriores calculará la frecuencia con la que aparece cada una de las palabras y visualizará el resultado.

La función *EstaPalabra* verificará si la palabra leída de la entrada estándar está en la matriz. Esta función devolverá un valor **true** si la palabra está en la matriz y **false** en caso contrario.

La función *InsertarPalabra* permitirá añadir una nueva palabra al final de la matriz.

La función *VisualizarPalabra* visualizará cada una de las palabras de la matriz y el número de veces que apareció.

4. Modificar el programa anterior para que la función *InsertarPalabra* inserte cada nueva palabra en el orden que le corresponde alfabéticamente, moviendo los elementos necesarios un lugar hacia atrás. De esta forma, cuando finalice la lectura del texto, la matriz estará ordenada.
5. Escribir un programa para evaluar la expresión $(ax + by)^n$. Para ello, tenga en cuenta las siguientes expresiones:

$$(ax + by)^n = \sum_{k=0}^n \binom{n}{k} (ax)^{n-k} (by)^k$$

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

$$n! = n * (n - 1) * (n - 2) * \dots * 2 * 1$$

- a) Escribir una función cuyo prototipo sea:

```
long factorial(int n);
```

La función *factorial* recibe como parámetro un entero y devuelve el factorial del mismo.

- b) Escribir una función con el prototipo:

```
long combinaciones(int n, int k);
```

La función *combinaciones* recibe como parámetros dos enteros n y k , y devuelve como resultado el valor de $\binom{n}{k}$.

- c) Escribir una función que tenga el prototipo:

```
double potencia(double base, int exponente);
```

La función *potencia* recibe como parámetros dos enteros, *base* y *exponente*, y devuelve como resultado el valor de $base^{\text{exponente}}$.

- d) La función **main** leerá los valores de a , b , n , x e y , y utilizando las funciones anteriores escribirá como resultado el valor de $(ax + by)^n$.

P A R T E

2

Mecanismos de abstracción

- Clases
- Operadores sobrecargados
- Clases derivadas
- Plantillas
- Excepciones
- Flujos

CAPÍTULO 9

© F.J.Ceballos/RA-MA

CLASES

Seguro que a estas alturas el término *clase* ya le es familiar. En los capítulos expuestos hasta ahora se han desarrollado aplicaciones sencillas y hemos utilizado algunos objetos de clases de la biblioteca de clases de C++. A partir de ahora, asumirá que un programa orientado a objetos sólo se compone de objetos y que un objeto es la concreción de una clase. Es hora pues de entrar con detalle en la programación orientada a objetos, la cual tiene un elemento básico: la *clase*.

DEFINICIÓN DE UNA CLASE

Una *clase* es un tipo definido por el usuario que describe los atributos y los métodos de los objetos que se crearán a partir de la misma. Los *atributos* definen el estado de un determinado objeto y los *métodos* son las operaciones que definen su comportamiento. Forman parte de estos métodos los *constructores*, que permiten iniciar un objeto, y los *destructores*, que permiten destruirlo. Los atributos y los métodos se denominan en general *miembros* de la clase.

La definición de una clase consta de dos partes: el *nombre de la clase* precedido por la palabra reservada **class** y el *cuerpo de la clase* encerrado entre llaves y seguido de un punto y coma. Esto es:

```
class nombre_clase
{
    cuerpo de la clase
};
```

El *cuerpo de la clase* en general consta de modificadores de acceso (**public**, **protected** y **private**), atributos, mensajes y métodos. Un método implícitamente define un mensaje (el nombre del método es el mensaje).

Por ejemplo, un círculo puede ser descrito por la posición (x, y) de su centro y por su *radio*. Hay varias cosas que nosotros podemos hacer con un círculo: calcular la longitud de la circunferencia, calcular el área del círculo, etc. Cada círculo es diferente (por ejemplo, tienen el centro o el radio diferente); pero visto como una *clase* de objetos, el círculo tiene propiedades intrínsecas que nosotros podemos agrupar en una definición. El siguiente ejemplo define la clase *Circulo*. Obsérvese cómo los atributos y los métodos forman el cuerpo de la clase.

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

class Circulo
{
    // miembros privados
private:
    double x, y;      // coordenadas del centro
    double radio;    // radio del círculo

    // miembros protegidos
protected:
    void msgEsNegativo()
    {
        cout << "El radio es negativo. Se convierte a positivo\n";
    }

    // miembros públicos
public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r) // constructor
    {
        x = cx; y = cy;
        if (r < 0)
        {
            msgEsNegativo();
            r = -r;
        }
        radio = r;
    }

    double longCircunferencia()
    {
        return 2 * 3.1415926 * radio;
    }

    double areaCirculo()
    {
        return 3.1415926 * radio * radio;
    }
};
```

Este ejemplo define un nuevo tipo de datos, *Circulo*, que puede ser utilizado dentro de un programa fuente exactamente igual que cualquier otro tipo. Un objeto de la clase *Circulo* tendrá los atributos *x*, *y* y *radio*, los métodos *msgEsNegativo*, *longCircunferencia* y *areaCirculo* y dos constructores *Circulo*, uno sin parámetros y otro con ellos.

Atributos

Los atributos constituyen la *estructura interna* de los objetos de una clase. En C++ un atributo también se denomina *dato miembro*. Para declarar un atributo, proceda exactamente igual que ha hecho para declarar cualquier otra variable en cualquier otra parte de un programa. Por ejemplo:

```
class Circulo
{
private:
    double x, y;
    double radio;
    // ...
};
```

En una clase, cada atributo debe tener un nombre único. En cambio, se puede utilizar el mismo nombre con atributos, y con miembros en general, que pertenezcan a diferentes clases, porque una clase define su propio ámbito.

No es posible asignar un valor inicial a un atributo de una clase (excepto si se declara entero, **static** y **const**). Por ejemplo, en la clase *Circulo* no podemos iniciar el radio con el valor 1, aunque generalmente esto no es necesario, ya que como expondremos un poco más adelante este tipo de operaciones son típicas del constructor de la clase:

```
class Circulo
{
private:
    double x, y;
    double radio = 1; // error: iniciación no permitida
    // ...
};
```

También podemos declarar como atributos de una clase objetos de otras clases existentes. El siguiente ejemplo define la clase *Punto* y después declara el atributo *centro*, de *Circulo*, de la clase *Punto*.

```
class Punto
{
```

```

private:
    double x, y;

public:
    Punto() {}
    Punto(double cx, double cy) { x = cx; y = cy; }
};

class Circulo
{
private:
    Punto centro; // coordenadas del centro
    double radio; // radio del círculo
    // ...
};

```

Observe ahora que la clase *Circulo* tiene un atributo *centro* de la clase *Punto*, lo que implica definir previamente la clase *Punto*.

Un objeto de una clase no puede ser atributo de ella misma, a no ser que se declare **static**, pero sí puede serlo un puntero al objeto. Por ejemplo:

```

class Circulo
{
private:
    // ...
    Circulo anterior; // error: objeto de la misma clase
    Circulo *panterior; // correcto
    // ...
};

```

Métodos de una clase

Los métodos generalmente forman lo que se denomina *interfaz* o medio de acceso a la estructura interna de los objetos; ellos definen las operaciones que se pueden realizar con sus atributos. En C++ un método de una clase también se denomina *función miembro* de la clase. Desde el punto de vista de la POO, el conjunto de todos estos métodos se corresponde con el conjunto de mensajes a los que los objetos de una clase pueden responder. Esto significa que los miembros de una clase sólo podrán ser accedidos por objetos de dicha clase.

Para definir un método de una clase, proceda exactamente igual que ha hecho para definir cualquier otro método (o función) en las aplicaciones realizadas en los capítulos anteriores. Recuerde también que los métodos no se pueden anidar. Como ejemplo puede observar los métodos *Circulo* y *longCircunferencia* de la clase *Circulo*.

```

class Circulo
{
    // ...
    public:
        Circulo(double cx, double cy, double r) // constructor
        {
            x = cx; y = cy;
            if (r < 0)
            {
                msgEsNegativo();
                r = -r;
            }
            radio = r;
        }

        double longCircunferencia()
        {
            return 2 * 3.1415926 * radio;
        }
        // ...
};

```

Control de acceso a los miembros de la clase

El concepto de clase incluye la idea de ocultación de datos, que básicamente consiste en que no se puede acceder directamente a los atributos de un objeto, sino que hay que hacerlo a través de métodos de su clase. Esto quiere decir que, de forma general, el usuario de la clase sólo tendrá acceso a uno o más métodos que le permitirán acceder a los miembros privados, ignorando la disposición de éstos (dichos métodos se denominan *métodos de acceso*). De esta forma se consiguen dos objetivos importantes:

1. Que el usuario no tenga acceso directo a la estructura de datos interna de la clase, para que no pueda generar código basado en esa estructura.
2. Que si en un momento determinado alteramos la definición de la clase, excepto el prototipo de los métodos, todo el código escrito por el usuario basado en estos métodos no tendrá que ser retocado.

Piense que si el objetivo uno no se cumpliera, cuando se diera el objetivo dos el usuario tendría que reescribir el código que hubiera desarrollado basándose en la estructura interna de los datos.

Para controlar el acceso a los miembros de una clase, C++ provee las palabras clave **private** (privado), **protected** (protegido) y **public** (público), aunque también es posible omitirlas, en cuyo caso el acceso se supone privado. Estas palabras

clave, denominadas *modificadores de acceso*, son utilizadas para indicar el tipo de acceso permitido a cada miembro de la clase. Si observamos la clase *Circulo* expuesta anteriormente, identificamos miembros privados, protegidos y públicos.

```
class Circulo
{
    // miembros privados
private:
    double x, y;    // coordenadas del centro
    double radio;  // radio del círculo

    // miembros protegidos
protected:
    void msgEsNegativo() { ... }
    // miembros públicos
public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r) { ... } // constructor
    double longCircunferencia() { ... }
    double areaCirculo() { ... }
};
```

Es importante no olvidar que un miembro de una clase sólo puede ser accedido, implícita o explícitamente, por un objeto de esa clase. En el ejemplo siguiente, el método *areaCirculo* de la clase *Circulo* es accedido por el objeto *c* de la misma clase; en POO se dice que el objeto *c* recibe el mensaje *areaCirculo* y responde ejecutando el método del mismo nombre.

```
int main()
{
    Circulo c(100, 200, 10); // invoca al constructor y construye c
    double area = c.areaCirculo(); // c recibe el mensaje areaCirculo
    double circ = longCircunferencia(); // error: no es una función
}                                     // global
```

Puede haber varias secciones privadas, protegidas o públicas. Cada una de ellas finaliza donde comienza la siguiente.

Acceso público

Un miembro de una clase declarado **public** (público) puede ser accedido por un objeto de esa clase en cualquier parte de la aplicación donde el objeto en cuestión sea accesible. Los miembros públicos de una clase constituyen la interfaz pública de los objetos de esa clase.

```
int main()
{
```

```
Circulo c(100, 200, 10);
double area = c.areaCirculo(); // correcto, miembro público
}
```

Una estructura (**struct**) es una clase cuyos miembros son públicos, por omisión.

Acceso privado

Un miembro de una clase declarado **private** (privado) puede ser accedido por un objeto de esa clase sólo desde los métodos de dicha clase (vea más adelante *El puntero implícito this*). Esto significa que no puede ser accedido por los métodos de cualquier otra clase, incluidas las subclases, ni por las funciones externas de la aplicación, como por ejemplo, la función **main**.

```
int main()
{
    Circulo c(100, 200, 10);
    double r = c.radio; // error: miembro privado
}
```

Acceso protegido

Un miembro de una clase declarado **protected** (protegido) se comporta exactamente igual que uno privado para las funciones externas o para los métodos de cualquier otra clase, pero actúa como un miembro público para los métodos de sus subclases.

Clases en ficheros de cabecera

En C++ un método de una clase es una definición que puede o no estar incluida en el cuerpo de la misma. Lo más común es que el cuerpo de una clase contenga solamente los atributos y los prototipos de sus métodos (parte que se conoce como declaración de la clase), razón por la que normalmente se escribe en un fichero de cabecera; de esta forma podrá ser incluida (**#include**) en cualquier fichero fuente que necesite utilizarla. Por ejemplo, volviendo a la clase *Circulo*, podríamos escribir la declaración de la misma en un fichero *circulo.h* así:

```
// circulo.h - Declaración de la clase Circulo
//
class Circulo
{
    // miembros privados
private:
    double x, y;    // coordenadas del centro
```

```
        double radio;    // radio del círculo

// miembros protegidos
protected:
    void msgEsNegativo();

// miembros públicos
public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r); // constructor
    double longCircunferencia();
    double areaCirculo();
};
```

Y las definiciones de los métodos, lo más común es escribirlas en otro fichero fuente con extensión *.cpp*, en vez de en el *.h*. De esta forma evitaremos por una parte que la definición de un mismo método tenga que ser traducida en los distintos ficheros *.cpp* que la incluyan y por otra, exponer el código fuente al usuario de la clase (podemos darle el fichero *.cpp* compilado más el correspondiente fichero de cabecera). Por ejemplo, continuando con la clase *Circulo*, podríamos escribir la definición de sus métodos en un fichero *circulo.cpp* así:

```
// circulo.cpp - Definición de los métodos de la clase Circulo
#include <iostream>
#include "circulo.h"

using namespace std;

void Circulo::msgEsNegativo()
{
    cout << "El radio es negativo. Se convierte a positivo\n";
}

Circulo::Circulo(double cx, double cy, double r) // constructor
{
    x = cx; y = cy;
    if (r < 0)
    {
        msgEsNegativo();
        r = -r;
    }
    radio = r;
}

double Circulo::longCircunferencia()
{
    return 2 * 3.1415926 * radio;
}
```



```
double Circulo::areaCirculo()
{
    return 3.1415926 * radio * radio;
}
```

Para que este fichero, denominado también unidad de traducción, pueda ser compilado satisfactoriamente debe incluir el fichero de cabecera *circulo.h*, puesto que el compilador para realizar la traducción de esta unidad necesita conocer los tipos y demás declaraciones a las que se hace referencia en la misma.

Se puede observar que para definir un método fuera del cuerpo de la clase, hay que indicar a qué clase pertenece dicho método; de lo contrario, el compilador interpretará que se trata de una función externa, en vez de un método de una clase. Para ello hay que especificar el nombre de la clase antes del nombre del método, separado del mismo por el operador de ámbito (::).

```
double Circulo::areaCirculo()
{
    return 3.1415926 * radio * radio;
}
```

Por otra parte, el hecho de especificar la clase a la que pertenece un método define al método dentro del ámbito de esa clase, lo que permite que existan métodos con el mismo nombre en diferentes clases. Por ejemplo:

```
void Punto::asignar(double d)
{
    // ...
}

void Circulo::asignar(double d)
{
    // ...
}
```

Un fichero de cabecera, además de la declaración de la clase, debe contener también los métodos **inline** de la misma, para que el compilador pueda acceder al código fuente de los mismos y reemplazar con él, si procede, cada una de las llamadas a los mismos.

Los métodos definidos en el cuerpo de la clase son por definición **inline**.

También debemos pensar que una misma unidad de traducción (un fichero *.cpp*) puede contener varias directrices **#include**, lo que puede dar lugar a incluir en esta unidad más de una copia de la declaración de una clase y/o de otras declaraciones; por ejemplo, simplemente porque un fichero de cabecera incluido expli-

citamente por el desarrollador sea a su vez incluido por otro fichero de cabecera, hecho que generalmente pasará desapercibido. Esto provocaría errores de redefinición durante la compilación de ese módulo. Para evitar este problema, cada fichero de cabecera debe contener las directrices siguientes:

```
#if !defined ( _NOMBRE_H_ )
#define _NOMBRE_H_

// contenido del fichero de cabecera (nombre.h)

#endif // _NOMBRE_H_
```

donde *NOMBRE*, generalmente, coincide con el nombre del fichero de cabecera.

Si aplicamos la teoría expuesta a la clase *Circulo*, vista anteriormente, obtendremos el siguiente resultado:

```
// circulo.h - Declaración de la clase Circulo
#if !defined( _CIRCULO_H_ )
#define _CIRCULO_H_

class Circulo
{
    // cuerpo de la clase
};

#endif // _CIRCULO_H_
```

Cuando un fichero de cabecera como el del ejemplo anterior es incluido en un módulo por primera vez, el símbolo *_NOMBRE_H_* no está definido; el preprocesador se dará cuenta de esto al evaluar la condición de la directriz **#if**; entonces, al ejecutar la directriz **#define**, lo define y, a continuación, incluye el fichero *nombre.h*. Si posteriormente tratamos de incluir el mismo fichero de cabecera, el símbolo *_NOMBRE_H_* ya está definido, lo que da lugar a que la condición de la directriz **#if** sea falsa y se ignore el contenido hasta **#endif**.

Posteriormente, para utilizar la clase *Circulo* simplemente tiene que incluir el fichero de cabecera *circulo.h* en los ficheros fuente donde se haga referencia a ella. Lógicamente, cuando compile su aplicación, tiene que enlazar con la misma el fichero *circulo.obj* (resultado de la compilación del fichero *circulo.cpp*) que contiene las definiciones de los métodos de la clase declarada en *circulo.h*. Tiene tres formas de enlazar este fichero: especificándolo directamente en la línea de órdenes; creando un fichero de proyecto (*.mak*) que incluya *circulo.obj* o *circulo.cpp* junto con los demás ficheros de la aplicación o incluyéndolo en una biblioteca que sería referenciada en el proceso de compilación y enlace. Esta forma de trabajar permite que un usuario necesite conocer solamente la interfaz de la clase,

sin importarle su implementación. Si posteriormente se modifica la clase *Circulo*, sin cambiar los prototipos de los métodos de la interfaz pública, lo único que tiene que hacer el usuario es volver a compilar su aplicación.

Como ejemplo, supongamos que escribimos el siguiente programa, que almacenamos en el fichero *areacir.cpp*:

```
// areacir.cpp - programa que utiliza la clase Circulo
#include <iostream>
#include "circulo.h"
using namespace std;

int main()
{
    Circulo c(100, 200, 10); // invoca al constructor y construye c
    double area = c.areaCirculo(); // c recibe el mensaje areaCirculo
    cout << area << endl;
}
```

Para realizar desde la línea de órdenes las fases de compilación y enlace de este programa tendríamos que escribir una orden análoga a la siguiente:

```
g++ areacir.cpp circulo.cpp -o areacir.exe
```

IMPLEMENTACIÓN DE UNA CLASE

La programación orientada a objetos con C++ sugiere escribir la declaración de cada clase en un fichero de cabecera y su definición en un fichero *.cpp*, fundamentalmente para reutilizar y mantener dicha clase posteriormente con facilidad. Como ejemplo, diseñaremos una clase que almacene una fecha, verificando que es correcta; esto es, que el día esté entre los límites 1 y días del mes, que el mes esté entre los límites 1 y 12 y que el año sea mayor o igual que 1582 (año gregoriano).

Parece lógico que la estructura de datos de un objeto fecha esté formada por los atributos *día*, *mes* y *año* y que permanezca oculta al usuario. Por otra parte, las operaciones sobre estos objetos tendrán que permitir asignar una fecha, método *asignarFecha*, obtener una fecha de un objeto existente, método *obtenerFecha*, y verificar si la fecha que se quiere asignar es correcta, método *fechaCorrecta*. Estos tres métodos formarán la interfaz pública. Cuando el día corresponda al mes de febrero, el método *fechaCorrecta* necesitará comprobar si el año es bisiesto, para lo que añadiremos el método *bisiesto*. Ya que un usuario no necesita acceder a este método, lo declararemos protegido con la intención de que, en un futuro, sí pueda ser accedido desde una subclase. Según lo expuesto, podemos escribir una clase denominada *CFecha* así:

```
// fecha.h - Declaración de la clase CFecha
class CFecha
{
    // Atributos
    private:
        int dia, mes, anyo;

    // Métodos
    protected:
        bool bisiestro();
    public:
        void asignarFecha(int dd, int mm, int aaaa);
        void obtenerFecha(int& dd, int& mm, int& aaaa) const;
        bool fechaCorrecta();
};
```

El paso siguiente es definir cada uno de los métodos. Al hablar de los modificadores de acceso quedó claro que cada uno de los métodos de una clase tiene acceso directo al resto de los miembros. Según esto, la definición del método *asignarFecha* puede escribirse así:

```
void CFecha::asignarFecha(int dd, int mm, int aaaa)
{
    dia = dd; mes = mm; anyo = aaaa;
}
```

Observe que, por ser *asignarFecha* un método de la clase *CFecha*, puede acceder directamente a los atributos *dia*, *mes* y *anyo* de su misma clase, independientemente de que sean privados. Estos atributos corresponderán en cada caso al objeto que recibe el mensaje *asignarFecha* (objeto para el que se invoca el método; vea más adelante, en este mismo capítulo, el puntero implícito **this**). Por ejemplo, si declaramos los objetos *fecha1* y *fecha2* de la clase *CFecha* y enviamos a *fecha1* el mensaje *asignarFecha*,

```
fecha1.asignarFecha(dd, mm, aaaa);
```

como respuesta a este mensaje, se ejecuta el método *asignarFecha* que asigna los datos *dd*, *mm* y *aaaa* al objeto *fecha1*; esto es, a *fecha1.dia*, *fecha1.mes* y *fecha1.anyo*; y si a *fecha2* le enviamos también el mensaje *asignarFecha*,

```
fecha2.asignarFecha(dd, mm, aaaa);
```

como respuesta a este mensaje, se ejecuta el método *asignarFecha* que asigna los datos *dd*, *mm* y *aaaa* al objeto *fecha2*; esto es, a *fecha2.dia*, *fecha2.mes* y *fecha2.anyo*.

Siguiendo las reglas enunciadas, finalizaremos el diseño de la clase escribiendo el resto de los métodos. El resultado que se obtendrá será la clase *CFecha* que se observa a continuación:

```
// fecha.cpp - Definición de los métodos de la clase CFecha
#include "fecha.h"

bool CFecha::bisiesto()
{
    return ((anyo % 4 == 0) && (anyo % 100 != 0) || (anyo % 400 == 0));
}

void CFecha::asignarFecha(int dd, int mm, int aaaa)
{
    dia = dd; mes = mm; anyo = aaaa;
}

void CFecha::obtenerFecha(int& dd, int& mm, int& aaaa) const
{
    *dd = dia; *mm = mes; *aaaa = anyo;
}

bool CFecha::fechaCorrecta()
{
    bool diaCorrecto, mesCorrecto, anyoCorrecto;

    anyoCorrecto = (anyo >= 1582); // ¿año correcto?
    mesCorrecto = (mes >= 1) && (mes <= 12); // ¿mes correcto?
    switch (mes)
    // ¿día correcto?
    {
        case 2:
            if (bisiesto())
                diaCorrecto = (dia >= 1 && dia <= 29);
            else
                diaCorrecto = (dia >= 1 && dia <= 28);
            break;
        case 4: case 6: case 9: case 11:
            diaCorrecto = (dia >= 1 && dia <= 30);
            break;
        default:
            diaCorrecto = (dia >= 1 && dia <= 31);
    }
    return diaCorrecto && mesCorrecto && anyoCorrecto;
}
```

Resumiendo: la funcionalidad de esta clase está soportada por los atributos privados *dia*, *mes* y *anyo* y por los métodos *asignarFecha*, *obtenerFecha*, *fechaCorrecta* y *bisiesto*.

El método público *asignarFecha* recibe tres enteros y los almacena en los atributos *día*, *mes* y *anyo* del objeto que recibe el mensaje *asignarFecha* (objeto para el que se invoca dicho método).

El método público *obtenerFecha* permite extraer los datos *día*, *mes* y *anyo* del objeto que recibe el mensaje *obtenerFecha*.

El método público *fechaCorrecta* verifica si la fecha que se desea asignar al objeto que recibe este mensaje es correcta. Este método devuelve **true** si la fecha es correcta y **false** en caso contrario.

El método protegido *bisiesto* verifica si el año de la fecha que se desea asignar al objeto que recibe este mensaje es bisiesto. Este método retorna **true** si el año es bisiesto y **false** en caso contrario.

MÉTODOS SOBRECARGADOS

En los capítulos anteriores, al trabajar con la biblioteca de C++ nos hemos encontrado con clases que implementan varias veces el mismo método. Por ejemplo, al hablar de la E/S dijimos que la clase **basic_istream** sobrecarga el operador `>>` con el fin de ofrecer al programador una notación cómoda que le permita obtener de la entrada estándar datos de cualquier tipo primitivo o derivado predefinido. Por ejemplo, algunas sobrecargas de este operador son:

```
basic_istream<...>& operator>>(int& n);
basic_istream<...>& operator>>(double& n);
basic_istream<...>& operator>>(basic_istream<...>& is, char *s);
```

También dijimos que la clase **basic_ostream** sobrecarga el operador `<<` con el fin de ofrecer al programador una notación cómoda que le permita enviar a la salida estándar datos de cualquier tipo primitivo o derivado predefinido. Por ejemplo:

```
basic_ostream<...>& operator<<(int n);
basic_ostream<...>& operator<<(double n);
basic_ostream<...>& operator<<(basic_ostream<...>& os, char *s);
```

¿En qué se diferencian estos métodos sobrecargados? En su número de parámetros y/o en el tipo de los mismos.

Pues bien, cuando en una clase un mismo método se define varias veces con distinto número de parámetros, o bien con el mismo número de parámetros pero diferenciándose una definición de otra en que al menos un parámetro es de un tipo diferente, se dice que el método está *sobrecargado*.

Los métodos sobrecargados pueden diferir también en el tipo del valor retornado. Ahora bien, el compilador C++ no admite que se declaren dos métodos que sólo difieran en el tipo del valor retornado; deben diferir también en la lista de parámetros; esto es, lo que importa son el número y el tipo de los parámetros.

La sobrecarga de métodos elimina la necesidad de nombrar de forma diferente métodos que en esencia hacen lo mismo, o también hace posible que un método se comporte de una u otra forma según el número de argumentos con el que sea invocado, como es el caso de los métodos **operator<<** y **operator>>**.

Como ejemplo, sobrecargaremos el método *asignarFecha* para que pueda ser invocado con cero argumentos; con un argumento, el día; con dos argumentos, el día y el mes; y con tres argumentos, el día, el mes y el año. Los datos día, mes o año omitidos en cualquiera de los casos serán obtenidos de la fecha actual proporcionada por el sistema.

La fecha actual del sistema se puede obtener a partir de las funciones **localtime** y **time** de la biblioteca de C. La función **localtime** utiliza una variable de tipo **static struct tm** para realizar la conversión, y lo que devuelve es la dirección de esa variable (vea el apéndice *La biblioteca de C*).

```
void CFecha::asignarFecha()
{
    // Asignar, por omisión, la fecha actual.
    struct tm *fh;
    time_t segundos;

    time(&segundos);
    fh = localtime(&segundos);

    dia = fh->tm_mday;          // día de 1 a 31
    mes = fh->tm_mon+1;        // mes de 0 a 11; enero = 0
    anyo = fh->tm_year+1900;    // año - 1900
}

void CFecha::asignarFecha(int dd)
{
    asignarFecha();
    dia = dd;
}

void CFecha::asignarFecha(int dd, int mm)
{
    asignarFecha();
    dia = dd; mes = mm;
}
```

```
void CFecha::asignarFecha(int dd, int mm, int aaaa)
{
    dia = dd; mes = mm; anyo = aaaa;
}
```

Como se puede observar, el que una definición del método invoque a otra es una técnica de método abreviado que da como resultado métodos más cortos.

Por cada llamada al método *asignarFecha* que escribamos en un programa, el compilador C++ debe resolver cuál de los métodos con el nombre *asignarFecha* es invocado. Esto lo hace comparando el número y tipos de los argumentos especificados en la llamada con los parámetros especificados en las distintas definiciones del método. El siguiente ejemplo muestra las posibles formas de invocar al método *asignarFecha*:

```
fecha.asignarFecha();
fecha.asignarFecha(dd);
fecha.asignarFecha(dd, mm);
fecha.asignarFecha(dd, mm, aaaa);
```

Si el compilador C++ no encontrara un método exactamente con los mismos tipos de argumentos especificados en la llamada, realizaría sobre dichos argumentos las conversiones implícitas permitidas entre tipos, tratando de adaptarlos a alguna de las definiciones existentes del método. Si este intento fracasa, entonces se producirá un error.

PARÁMETROS CON VALORES POR OMISIÓN

En ocasiones, nos podremos ahorrar escribir múltiples formas de un mismo método, si utilizamos parámetros con valores, por omisión. Esto ocurrirá cuando partiendo de un método con una lista de n parámetros preestablecidos, tengamos la necesidad de pasar 0 a n argumentos. El ejemplo anterior se ajusta perfectamente a lo expuesto. En este caso, podemos sustituir todas las formas anteriores de *asignarFecha* por una única forma del método cuyo prototipo sea, por ejemplo, así:

```
void asignarFecha(int dd = 0, int mm = 0, int aaaa = 0);
```

La definición de este método será como se indica a continuación:

```
void CFecha::asignarFecha(int dd, int mm, int aaaa)
{
    struct tm *fh;
    time_t segundos;

    time(&segundos);
```



```

    fh = localtime(&segundos); // fecha actual
    // Por omisión, se asigna el día, mes o año actual
    if (dd) dia = dd; else dia = fh->tm_mday;
    if (mm) mes = mm; else mes = fh->tm_mon+1;
    if (aaaa) anyo = aaaa; else anyo = fh->tm_year+1900;
}

```

IMPLEMENTACIÓN DE UNA APLICACIÓN

Una aplicación orientada a objetos consiste en una o más clases y funciones externas, de las cuales una tiene que ser la función **main**. Por ejemplo, para comprobar que la clase *CFecha* que acabamos de diseñar trabaja correctamente, podemos escribir una aplicación *test* según se muestra a continuación:

```

// test.cpp - Trabajar con la clase CFecha
#include <iostream>
#include "fecha.h"
using namespace std;

void leerFecha(int&, int&, int&);
void visualizarFecha(const CFecha& fecha);

int main()
{
    CFecha fecha; // objeto de tipo CFecha
    int dd = 0, mm = 0, aaaa = 0;
    do
    {
        leerFecha(dd, mm, aaaa);
        fecha.asignarFecha(dd, mm, aaaa);
    }
    while (!fecha.fechaCorrecta());
    visualizarFecha(fecha);
}

void leerFecha(int& dia, int& mes, int& anyo)
{
    cout << "día: "; cin >> dia;
    cout << "mes: "; cin >> mes;
    cout << "año: "; cin >> anyo;
}

void visualizarFecha(const CFecha& fecha)
{
    int dd, mm, aaaa;

    fecha.obtenerFecha(dd, mm, aaaa);
    cout << dd << "/" << mm << "/" << aaaa << "\n";
}

```

Notar que la clase *CFecha* declara los atributos *día*, *mes* y *año* privados. Esto quiere decir que sólo son accesibles por los métodos de su clase. Si un método de otra clase intenta acceder a uno de estos atributos, el compilador genera un error. Por ejemplo:

```
int main()
{
    CFecha fecha; // objeto de tipo CFecha
    int dd = 0, mm = 0, aaaa = 0;
    // ...
    int dd = fecha.dia; // error: día es un miembro privado
    fecha.mes = 1;      // error: mes es un miembro privado
}
```

En cambio, los métodos *asignarFecha*, *obtenerFecha* y *fechaCorrecta* son públicos. Por lo tanto, son accesibles, además de por los métodos de su clase, por cualquier otro método de otra clase o función externa. Sirva como ejemplo la función *visualizarFecha* de la aplicación *test*. Esta función presenta en la salida estándar la fecha almacenada en el objeto que se le pasa como argumento. Observe que tiene que invocar al método *obtenerFecha* para acceder a los datos de un objeto *CFecha*. Esto es así porque un método que no es miembro de la clase del objeto no tiene acceso a sus datos privados.

La función externa *leerFecha* obtiene de la entrada estándar la fecha que se desea almacenar en un objeto a través del método *asignarFecha*.

EL PUNTERO IMPLÍCITO **this**

Cada objeto mantiene su propia copia de los atributos pero no de los métodos de su clase, de los cuales sólo existe una copia para todos los objetos de esa clase. Esto es, cada objeto almacena sus propios datos, pero para acceder y operar con ellos, todos comparten los mismos métodos definidos en su clase. Por lo tanto, para que un método conozca la identidad del objeto particular para el que ha sido invocado, C++ proporciona un puntero al objeto denominado **this**. Así, por ejemplo, si creamos un objeto *fecha1* y a continuación le enviamos el mensaje *asignarFecha*,

```
fecha1.asignarFecha(dia, mes, anyo);
```

C++ define el puntero **this** para permitir referirse al objeto *fecha1* en el cuerpo del método que se ejecuta como respuesta al mensaje. Esa definición es así:

```
CFecha *const this = &fecha1;
```

Y cuando realizamos la misma operación con otro objeto *fecha2*,

```
fecha2.asignarFecha(dia, mes, anyo);
```

C++ define de nuevo el puntero **this**, para referirse al objeto *fecha2*, de la forma:

```
CFecha *const this = &fecha2;
```

Según lo expuesto, el método *asignarFecha* podría ser definido también como se muestra a continuación:

```
void CFecha::asignarFecha(int dd, int mm, int aaaa)
{
    this->dia = dd; this->mes = mm; this->anyo = aaaa;
    // O bien:
    // (*this).dia = dd; (*this).mes = mm; (*this).anyo = aaaa;
}
```

¿Qué representa **this** en este método? Según lo explicado, **this** es un puntero al objeto (***this**) que recibió el mensaje *asignarFecha*; esto es, al objeto sobre el que se está realizando el proceso llevado a cabo por el método *asignarFecha*.

Observe ahora la función **main** de la aplicación *test* presentada anteriormente. En ella hemos declarado un objeto *fecha* de la clase *CFecha* y posteriormente le hemos enviado un mensaje *fechaCorrecta*:

```
do
{
    leerFecha(dd, mm, aaaa);
    fecha.asignarFecha(dd, mm, aaaa);
}
while (!fecha.fechaCorrecta());
```

En este caso, igual que en el ejemplo anterior, el método *fechaCorrecta* conoce con exactitud el objeto sobre el que tiene que actuar, puesto que se ha expresado explícitamente. Pero, ¿qué pasa con el método *bisiesto* que se encuentra sin referencia directa alguna en el cuerpo del método *fechaCorrecta*?

```
boolean CFecha::fechaCorrecta()
{
    // ...
    if (bisiesto())
    // ...
}
```

En este otro caso, la llamada no es explícita como en el caso anterior. Lo que ocurre en la realidad es que todas las referencias a los atributos y métodos del ob-

jeto para el que se invocó el método *fechaCorrecta* (objeto que recibió el mensaje *fechaCorrecta*) son implícitamente realizadas a través de **this**. Según esto, la sentencia **if** anterior podría escribirse también así:

```
if (this->bisiesto())
```

Normalmente, en un método no es necesario utilizar esta referencia para acceder a los miembros del objeto implícito, pero es útil cuando haya que devolver dicho objeto (***this**) o una referencia al mismo.

MÉTODOS Y OBJETOS CONSTANTES

Declarar un objeto **const** hace que cualquier intento accidental de modificar dicho objeto sea detectado durante la compilación, en vez de causar errores durante la ejecución.

Si declaramos explícitamente un objeto constante, es un error que el método invocado no sea también constante cuando se envía un mensaje a este objeto. Por ejemplo, si declaramos un objeto *cumpleaños* de la clase *CFecha* constante y le enviamos el mensaje *fechaCorrecta*,

```
const CFecha cumpleaños;
cumpleaños.fechaCorrecta();
```

el método *fechaCorrecta* tiene obligatoriamente que declararse y definirse constante (**const**); esto es:

```
class CFecha
{
    // ...
    int fechaCorrecta() const; // fechaCorrecta se declara constante
};

// fechaCorrecta se define constante
int CFecha::fechaCorrecta() const
{
    // cuerpo del método
}
```

En este ejemplo, se ha declarado constante el método *fechaCorrecta*. Cuando un método es constante, el objeto referenciado por el puntero **this** se asume constante. Esto supone que el puntero **this** quede implícitamente declarado constante a un objeto constante; esto es:

```
const CFecha *const this;
```

Por lo tanto, un método declarado constante no puede modificar la representación interna de los objetos para los que es invocado. Una forma de saltar este sistema de protección sería realizar una conversión explícita (**const_cast**) sobre **this** para que apuntara a un objeto no constante, pero esto no se considera un buen estilo de programación. Una mejor solución es declarar el atributo **mutable**, para que permita la actualización, independientemente de que sea un atributo de un objeto **const**.

```
class CFecha
{
    // Atributos
private:
    mutable int dia; // permitir la actualización en objetos const
    int mes, año;
    // Métodos
    // ...
};
```

Siguiendo con el ejemplo, como todas las referencias a los atributos y métodos de una clase en el cuerpo de otro método de la misma clase se hacen a través del puntero implícito **this**, si el método *fechaCorrecta* ha sido declarado constante, ¿qué pasa con el método *bisiesto* que se invoca desde el método *fechaCorrecta* y no ha sido declarado constante? Estamos otra vez en el caso de un objeto constante invocando a un método no constante. Una buena solución a este problema es declarar el método *bisiesto* también constante, ya que este método no necesita modificar el objeto para el que es invocado. Esto es:

```
class CFecha
{
    // ...
    int bisiesto() const;
    // ...
};

int CFecha::bisiesto() const
{
    // Cuerpo del método
}
```

Sin embargo, cuando un objeto no es constante, el método invocado para trabajar sobre él puede ser no constante o constante.

Por otra parte, debemos conocer que dos versiones de un mismo método que sólo difieran en **const** son sobrecargas de ese método (difieren en el tipo del parámetro implícito **this**). Por ejemplo, los dos prototipos siguientes indican que el método *bisiesto* está sobrecargado:

```
int bisiesto(); // versión que se ejecutará para objetos no const
int bisiesto() const; // versión que se ejecutará para objetos const
```

Finalmente, decir que un método declarado **const** no puede devolver una referencia a un objeto no **const**. Por ejemplo:

```
int& CFecha::obtenerDia() const
{
    return dia; // error: no se puede convertir 'const int' a 'int &'
}
```

Cuando se compila el método anterior, se muestra un error indicando que *dia* es constante, razón por la que no se le puede asociar una referencia a un objeto no constante, lo cual es lógico. Si recordamos, un método que devuelve una referencia puede utilizarse a la izquierda del operador de asignación, ya que actúa como un sinónimo del objeto retornado. Entonces, si se permitiera la acción anterior, se estaría permitiendo modificar un objeto constante:

```
fecha.obtenerDia() = dd;
```

En conclusión, el método anterior tendría que retornar bien el objeto (una copia), o bien una referencia a un objeto constante como se muestra a continuación:

```
const int& CFecha::obtenerDia() const
{
    return dia;
}
```

Como ejercicio, añadir los métodos *obtenerMes* y *obtenerAnyo*.

INICIACIÓN DE UN OBJETO

Sabemos que un objeto consta de una estructura interna (los atributos) y de una interfaz que permite acceder y manipular tal estructura (los métodos). Ahora, ¿cómo se construye un objeto de una clase cualquiera? Pues, de forma análoga a como se construye cualquier otra variable de un tipo predefinido. Por ejemplo:

```
int edad;
```

Este ejemplo define la variable *edad* del tipo predefinido **int**. En este caso, el compilador automáticamente reserva memoria para su ubicación, le asigna un valor (0 si se trata de una variable global o indeterminado si es local a un método) y procederá a su destrucción, cuando el flujo de ejecución vaya fuera del ámbito donde haya sido definida.

Esto nos hace pensar en la idea de que de alguna manera el compilador llama a un método de iniciación, *constructor*, para iniciar cada una de las variables declaradas, y a un método de eliminación, *destructor*, para liberar el espacio ocupado por dichas variables, justo al salir del ámbito en el que han sido definidas.

Pues bien, con un objeto de una clase ocurre lo mismo. Por ejemplo:

```
CFecha fecha;
```

Con objetos, el compilador proporciona un *constructor* público, por omisión, para cada clase definida (se entiende, por omisión del que escribe la clase; si éste no escribe un constructor, lo aporta el compilador). Este constructor será ejecutado después de que el propio programa, secuencial y recursivamente (un atributo de una clase puede ser iniciado con un objeto de otra clase), asigne memoria para cada uno de los atributos y los inicie. Igualmente, el compilador proporciona para cada clase de objetos un *destructor* público, por omisión, que será invocado justo antes de que se destruya un objeto con el fin de permitir realizar tareas de limpieza y liberar recursos.

¿Cómo implementa C++ el constructor y el destructor, por omisión, en una clase? Veámoslo en la clase *CFecha*:

```
class CFecha
{
    // Atributos
    private:
        int dia, mes, anyo;

    // Métodos
    protected:
        bool bisiestro();
        bool bisiestro() const;
    public:
        CFecha() {} // constructor, por omisión
        ~CFecha() {} // destructor, por omisión
        void asignarFecha(int dd = 1, int mm = 1, int aaaa = 1900);
        void obtenerFecha(int& dd, int& mm, int& aaaa) const;
        bool fechaCorrecta() const;
        const int& obtenerDia() const;
};
```

No obstante, como veremos a continuación, cuando el constructor proporcionado, por omisión, por C++ no satisfaga las necesidades de nuestra clase de objetos, podemos definir uno. Ídem para el destructor.

Constructor

En C++, una forma de asegurar que los objetos siempre contengan valores válidos es escribir un constructor. Un *constructor* es un método especial de una clase que es llamado automáticamente siempre que se crea un objeto de la misma, cosa que ocurre siempre que se declare una variable de esa clase. Su función es iniciar los nuevos objetos. Cuando se crea un objeto, C++ hace lo siguiente:

- Asigna memoria para el objeto.
- Inicia los atributos de ese objeto con los valores predeterminados por el sistema.
- Llama al constructor de la clase que puede ser uno entre varios, según se expone a continuación.

Dado que los constructores son métodos, admiten parámetros igual que éstos. Cuando en una clase no escribimos ningún constructor, el compilador añade uno público, por omisión, sin parámetros, según se vio en el apartado anterior.

```
CFecha() { /* Sin código */ }
```

Un *constructor por omisión* de una clase *C* es un constructor sin parámetros que no hace nada. Sin embargo, es necesario porque, según lo que acabamos de exponer, será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema.

Un *constructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece (por ejemplo, el constructor para la clase *CFecha* se denomina también *CFecha*), no se hereda, no puede retornar un valor (incluyendo **void**) y no puede ser declarado **const**, **static** o **virtual** (el primer modificador ya es conocido; los otros lo serán en la medida que ampliemos nuestros conocimientos sobre C++).

Como ejemplo, vamos a añadir un constructor a la clase *CFecha* con el fin de poder iniciar los atributos de cada nuevo objeto con unos valores determinados:

```
// fecha.h - Declaración de la clase CFecha
class CFecha
{
    // Atributos
    private:
        int dia, mes, anyo;

    // Métodos
```



```

public:
    CFecha(int dd, int mm, int aaaa); // constructor
    // ...
};

```

Observe que el constructor, salvo en casos excepcionales, debe declararse siempre público para que pueda ser invocado desde cualquier parte de una aplicación donde se cree un objeto de su clase.

```

// fecha.cpp - Definición de los métodos de la clase CFecha
#include <iostream>
#include "ctime"
#include "fecha.h"
using namespace std;

CFecha::CFecha(int dd, int mm, int aaaa) // constructor
{
    día = dd; mes = mm; anyo = aaaa;
    if (!fechaCorrecta())
    {
        cout << "Fecha incorrecta. Se asigna una fecha, por omisión.\n";
        día = 1; mes = 1; anyo = 2001;
    }
}
// ...

```

Cuando una clase tiene un constructor, éste será invocado automáticamente siempre que se cree un nuevo objeto de esa clase. El objeto se considera construido con los valores predeterminados por el sistema justo antes de iniciarse la ejecución del constructor. Por lo tanto, a continuación, desde el cuerpo del constructor es posible asignar valores a sus atributos, invocar a los métodos de su clase o bien llamar a métodos de otros objetos.

En el caso de que el constructor tenga parámetros, para crear un nuevo objeto hay que especificar la lista de argumentos correspondiente entre los paréntesis que siguen al nombre de la clase del objeto. El siguiente ejemplo muestra esto con claridad:

```

int main()
{
    // La siguiente línea invoca al constructor de la clase CFecha
    CFecha fecha(1, 3, 2012); // crear fecha iniciado con 1 3 2012
    visualizarFecha(fecha);
}

```

Este ejemplo define un objeto *fecha* e inicia sus atributos *día*, *mes* y *anyo* con los valores 1, 3 y 2012, respectivamente. Para ello, invoca al constructor *CFe-*

cha(int dd, int mm, int aaaa), le pasa los argumentos 1, 3 y 2012 y ejecuta el código que se especifica en el cuerpo del mismo. Una vez construido el objeto, visualizamos su contenido invocando a la función *visualizarFecha*, que a su vez invoca al método *obtenerFecha* de dicho objeto para obtener sus atributos. La siguiente línea es la salida del ejemplo anterior:

```
1/3/2012
```

Añadamos ahora a la función **main** del programa *test* del ejemplo anterior la línea de código que se indica a continuación. ¿Qué ocurrirá?

```
CFecha fecha1;
```

Quizás se sorprenda cuando el compilador C++ le indique que la clase *CFecha* no tiene ningún constructor sin parámetros, cuando anteriormente habíamos dicho que C++ proporciona para toda clase uno. Lo que sucede es que siempre que en una clase se define explícitamente un constructor, el constructor implícito (constructor por omisión) es reemplazado por éste.

Según lo expuesto, la definición explícita del constructor con parámetros *CFecha(int dd, int mm, int aaaa)* ha sustituido al constructor que C++ añadió a esa clase por omisión. Para solucionar este problema, hay que añadir a la clase un constructor público sin parámetros. Por ejemplo, el siguiente:

```
CFecha() { /* Sin código */ }
```

El constructor anterior realiza la misma función que el constructor por omisión. No obstante, en el caso de la clase *CFecha*, quizás sea más conveniente añadir un constructor con parámetros con valores por omisión:

```
CFecha(int dd = 1, int mm = 1, int aaaa = 2001); // constructor
```

Ahora, podemos invocar al constructor *CFecha* con 0, 1, 2 ó 3 argumentos, según se puede observar en las líneas de código siguientes:

```
CFecha fecha1;           // invoca al constructor sin argumentos
CFecha fecha2(3);       // invoca al constructor con 1 argumento
CFecha fecha3(15, 3);   // invoca al constructor con 2 argumentos
CFecha fecha4(1, 3, 2012); // invoca al constructor con 3 argumentos
```

De esta forma, con un solo constructor quedan resueltos todos los problemas planteados anteriormente. Pero, ¿qué diferencia hay entre este constructor y el constructor implícito por omisión? En ambos casos se invoca a los constructores, por omisión, para cada uno de los atributos; esto es, se construyen las variables *dia*, *mes* y *año* de tipo **int**, iniciándolas como corresponda, dependiendo de que

el objeto sea local o global. Pero en el caso del constructor explícito, se ejecuta a continuación el cuerpo del mismo que, según el ejemplo, asigna a cada variable un valor específico; esto supone una segunda asignación que se traduce en más tiempo de ejecución. Lo ideal sería que este constructor pasara a los constructores de cada uno de los atributos los valores especificados para ser asignados durante su construcción, y que después se ejecutara el cuerpo del constructor *CFecha* para las operaciones adicionales, si las hubiera. Para esto, C++ proporciona la siguiente sintaxis:

```
CFecha::CFecha(int dd, int mm, int aaaa) : /* constructor */
    día(dd), mes(mm), anyo(aaaa)
{
    if (!fechaCorrecta())
    {
        cout << "Fecha incorrecta. Se asigna una fecha, por omisión.\n";
        día = 1; mes = 1; anyo = 2001;
    }
}
```

Los dos puntos a continuación de la lista de parámetros del constructor *CFecha* indican que sigue una lista de iniciadores; en este caso, de los atributos de la clase. Incluso, cuando no se requiera ninguna operación adicional, el cuerpo del constructor aparecerá vacío.

Se puede crear un objeto de cualquiera de las formas siguientes:

- Declarando un objeto global:

```
CFecha f; // se supone que f está declarada fuera de todo bloque
```

Obsérvese que escribimos *CFecha f*; y no *CFecha f()*. En este último caso el compilador asumiría que se trata de la declaración de una función, *f*, sin parámetros, que devuelve un objeto *CFecha*.

- Declarando un objeto local u objeto temporal:

```
void fx(int d, int m, int a)
{
    CFecha f(d, m, a); // equivale a: CFecha f = CFecha(d, m, a);
    // ...
}
```

- Invocando al operador **new** (objeto dinámico):

```
CFecha *fx()
{
```

```

    CFecha *p = new CFecha; // equivale a: CFecha *p = new CFecha();
    // ...
}

```

- Llamando explícitamente a un constructor:

```

CFecha fx()
{
    // ...
    return CFecha(d, m, a);
}

```

Asignación de objetos

Otra forma de iniciar un objeto es utilizando el operador de asignación (=). Por ejemplo:

```

CFecha fecha1(1, 3, 2012);
CFecha fecha2;
fecha2 = fecha1;

```

Este ejemplo crea los objetos *fecha1* y *fecha2* y, a continuación, asigna el contenido de *fecha1* a *fecha2*. Obsérvese que cuando se realiza la operación de asignación, ambos objetos existen.

Pruebe las operaciones anteriores y observe que todo funciona correctamente. Esto demuestra que el compilador C++ proporciona para cada clase un operador de asignación, por omisión. Se trata de un método público resultado de sobrecargar el operador =, que asigna uno a uno los atributos de un objeto a otro. Por ejemplo, el operador de asignación, por omisión, de la clase *CFecha* es así:

```

CFecha& CFecha::operator=(const CFecha& fecha)
{
    dia = fecha.dia;
    mes = fecha.mes;
    anyo = fecha.anyo;
    return *this;
}

```

Cuando el operador de asignación, por omisión, no sea adecuado por no ajustarse a lo que esperamos de él, escribiremos nuestra propia versión.

¿Qué prototipo tiene el operador de asignación? Como hemos dicho, se trata de una sobrecarga del operador = (**operator=**) que tiene un parámetro que es una referencia a un objeto constante de su clase, el objeto a copiar. El declarar el objeto constante impide que el objeto pasado se modifique, característica implícita

cuando el argumento se pasa por valor. Retorna una referencia al objeto resultante, lo cual permite realizar asignaciones encadenadas (por ejemplo, $a = b = c$). Si no devolviera nada (**void**), sólo se podrían realizar asignaciones simples. ¿Por qué? Veamos. Está claro que las dos sentencias siguientes realizan la misma operación:

```
b = c;           // llamada implícita al método operator=
b.operator=(c); // llamada explícita al método operator=
```

Si *operator=* devuelve una referencia al objeto resultante de la copia, este objeto puede, a su vez, ser copiado en otro ($a = b.operator=(c)$), operación que no se podría hacer si no devolviera nada (no olvide que la precedencia de los operadores de asignación es de derecha a izquierda). Según lo explicado, las dos sentencias siguientes realizan la misma operación:

```
a = b = c;
a.operator=(b.operator=(c));
```

El hecho de que el operador de asignación devuelva una referencia al objeto y no el propio objeto es simplemente por una cuestión de eficacia; esto es, de esta forma se evita una llamada al constructor copia (que estudiamos a continuación) para copiar el objeto devuelto en otro temporal y una llamada al destructor para eliminar el objeto temporal una vez realizada la asignación. Por la misma razón pasamos el parámetro por referencia.

Constructor copia

Otra forma de iniciar un objeto es asignándole otro objeto de su misma clase en el momento de su creación. Lógicamente, si se crea un objeto tiene que intervenir un constructor, que recibirá como único argumento el objeto con el que se iniciará; este constructor recibe el nombre de *constructor copia*. Por ejemplo:

```
CFecha fecha1(1, 3, 2012);
CFecha fecha2(fecha1); // crear fecha2 iniciado con fecha1
```

Este ejemplo crea primero el objeto *fecha1* y después *fecha2* iniciado con el contenido de *fecha1*. La sentencia anterior puede escribirse también así:

```
CFecha fecha2 = fecha1; // llama al constructor copia
```

Pruebe las operaciones anteriores y observe que todo funciona correctamente. Esto demuestra que el compilador C++ proporciona para cada clase un constructor público por omisión, que recibe el nombre de *constructor copia*, que permite

construir un objeto a partir de otro existente que recibe como parámetro. Por ejemplo, el constructor copia, por omisión, de la clase *CFecha* es así:

```
CFecha::CFecha(const CFecha& fecha) : /* constructor copia */  
    día(fecha.día), mes(fecha.mes), anyo(fecha.anyo)  
{  
}
```

En este caso, y a diferencia del operador de asignación, es necesario pasar el parámetro por referencia, porque pasarlo por valor implicaría una recursividad infinita (pasar un objeto por valor implica llamar al constructor copia).

Si analizamos las operaciones que realizan el operador de asignación y el constructor copia, llegaremos a la conclusión de que son las mismas, excepto en que el constructor copia no retorna nada. Basándonos en este hecho, otra forma de escribir el constructor copia sería invocando al operador de asignación, forma que resulta especialmente útil cuando el cuerpo del constructor sea extenso en código:

```
CFecha::CFecha(const CFecha& fecha) /* constructor copia */  
{  
    *this = fecha; // invoca al operador de asignación  
}
```

Cuando el constructor copia por omisión no sea adecuado por no ajustarse a lo que esperamos de él, escribiremos nuestra propia versión, lo que implicará escribir también nuestra propia versión del operador de asignación.

Tanto en el constructor copia como en el operador de asignación, no declarar su único parámetro **const** supone añadir una sobrecarga a este método. Esto es, el prototipo que se muestra a continuación es una sobrecarga del constructor copia expuesto anteriormente, por lo tanto, ambos pueden coexistir.

```
CFecha::CFecha(CFecha& fecha);
```

DESTRUCCIÓN DE OBJETOS

De la misma forma que existe un método que se ejecuta automáticamente cada vez que se construye un objeto, también existe un método que se invoca automáticamente cada vez que se destruye. Este método recibe el nombre de *destructor*.

Un objeto es destruido automáticamente al salir del ámbito en el que ha sido definido el objeto. Sin embargo, hay una excepción: los objetos creados dinámicamente por el operador **new** tienen que ser destruidos utilizando el operador **de-**

lete, de lo contrario el sistema destruiría la variable puntero pero no liberaría el espacio de memoria referenciado por ella.

Destructor

Un *destructor* es un método especial de una clase que se ejecuta antes de que un objeto de esa clase sea eliminado físicamente de la memoria. Un *destructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece precedido por una tilde `~`. Un *destructor* no es heredado, no tiene argumentos, no puede retornar un valor (incluyendo **void**) y no puede ser declarado **const** ni **static**, pero sí puede ser declarado **virtual**. Utilizando destructores virtuales podremos destruir objetos sin conocer su tipo (más adelante trataremos el mecanismo de los métodos virtuales).

Cuando en una clase no especificamos un destructor, el compilador añade uno público, por omisión. Por ejemplo, el destructor para la clase *CFecha* es declarado por el compilador C++ así:

```
~CFecha() {};
```

Para definir un destructor en una clase tiene que reescribir el método anterior. A diferencia de lo que ocurría con los constructores, en una clase sólo es posible definir un *destructor* (lógico, al no tener parámetros no se puede sobrecargar). En el cuerpo del mismo puede escribir cualquier operación que quiera realizar relacionada con el objeto que se vaya a destruir.

Resumiendo: un destructor es invocado automáticamente justo antes de que el objeto sea eliminado físicamente de la memoria. Y, ¿cuándo ocurre esto? Si es global o automático, cuando el flujo de ejecución sale fuera de su ámbito, y si ha sido creado dinámicamente, cuando apliquemos el operador **delete** sobre él.

Como ejemplo vamos a añadir a la clase *CFecha* un destructor para que simplemente muestre un mensaje cada vez que se destruya un objeto de la misma:

```
// fecha.h - Declaración de la clase CFecha
class CFecha
{
    // ...
public:
    CFecha(int dd = 1, int mm = 1, int aaaa = 2001); // constructor
    ~CFecha(); // destructor
    // ...
};
// ...
```

```

CFecha::~CFecha() // destructor
{
    cout << "Objeto CFecha destruido\n";
}
// ...

```

Ejecute ahora la aplicación *Test* con la función **main** que se muestra a continuación y observe los resultados.

```

int main()
{
    CFecha fecha1(1, 3, 2012); // crear e iniciar el objeto fecha1
    CFecha *pfecha2 = new CFecha(fecha1); // llama al constr. copia

    visualizarFecha(*pfecha2);
    delete pfecha2; // llama al destructor
}

```

Analizando este ejemplo, se observa que en la función **main** se crean dos objetos: uno automático y otro dinámico. Por lo tanto, una vez finalizada la ejecución de la aplicación, se podrá observar que se habrá visualizado el mensaje “Objeto CFecha destruido” tantas veces como objetos hay. Fijarse que para destruir un objeto creado dinámicamente hay que utilizar el operador **delete**. El operador **delete** libera la memoria asignada por **new** con el fin de destruir el objeto, por eso justo antes de esta operación, se invoca al destructor de la clase del objeto.

Si una clase tiene atributos que son objetos de otras clases, su destructor se ejecuta antes que los destructores de los atributos. En otras palabras, el orden de destrucción es inverso al orden de construcción.

Un destructor también se puede llamar explícitamente utilizando su nombre completo, según muestra la siguiente notación, aunque sólo en circunstancias muy poco habituales, es necesario realizar esta operación:

```

objeto.nombre_clase::~~nombre_clase();
pobjeto->nombre_clase::~~nombre_clase();

```

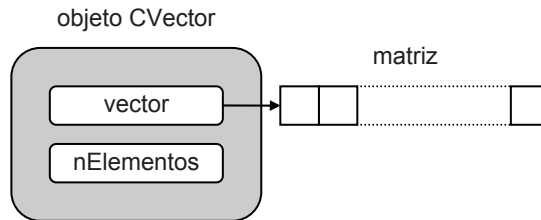
Generalmente se incluya el nombre de la clase a la que pertenece el destructor para que la tilde (~) no sea interpretada como el operador complemento a 1.

PUNTEROS COMO ATRIBUTOS DE UNA CLASE

Un atributo de una clase que sea un puntero requiere, generalmente, de una asignación de memoria, proceso que normalmente realizará el constructor. Sucede entonces que el espacio de memoria asignado es referenciado desde el objeto pero,

lógicamente, no pertenece al objeto, lo que puede dar lugar a problemas si no se implementan adecuadamente el constructor copia, el operador de asignación y el destructor, fundamentalmente.

Para ver lo expuesto con detalle, vamos a escribir una clase *CVector* para construir objetos que representen matrices numéricas con un número cualquiera de elementos. Por lo tanto, sería inapropiado definir como miembro privado de la clase *CVector* una matriz con un número fijo de elementos. En su lugar, definiremos un puntero, *vector*, a una matriz de tipo **double**, por ejemplo, para después asignar dinámicamente la cantidad de memoria necesaria para la matriz.



Según lo expuesto, la funcionalidad de la clase *CVector* estará soportada por los atributos:

- *vector*: una referencia a una matriz de valores de tipo **double**.
- *nElementos*: número de elementos de dicha matriz.

```
class CVector
{
    private:
        double *vector;
        int nElementos;

        // ...
};
```

y por los métodos:

- *constructores* para crear un objeto *CVector* con un número de elementos pre-determinado, con un número de elementos especificado, a partir de una matriz unidimensional, o bien a partir de otro objeto *CVector*.

El trabajo que tienen que realizar los constructores de la clase *CVector*, dependiendo de los casos, es asignar la memoria necesaria para la matriz de datos e iniciar dicha matriz con ceros (iniciación por omisión), con otra matriz o con otro vector, como podemos ver a continuación:

```
// Crear una matriz con 10 elementos por omisión
CVector::CVector()
{
    nElementos = 10;
    vector = asignarMem(nElementos);
    fill(vector, vector + nElementos, 0);
}

// Crear una matriz con ne elementos
CVector::CVector(int ne)
{
    if (ne < 1)
    {
        cerr << "Nº de elementos no válido: " << ne << "\n";
        return;
    }
    nElementos = ne;
    vector = asignarMem(nElementos);
    fill(vector, vector + nElementos, 0);
}

// Crear una matriz a partir de otra matriz primitiva
CVector::CVector(double *a, int ne)
{
    nElementos = ne;
    vector = asignarMem(nElementos);
    // Copiar los elementos de la matriz a
    for (int i = 0; i < nElementos; i++)
        vector[i] = a[i];
}

// Constructor copia
CVector::CVector(const CVector& v)
{
    nElementos = v.nElementos;
    vector = asignarMem(nElementos);
    // Copiar el objeto v
    for (int i = 0; i < nElementos; i++)
        vector[i] = v.vector[i];
}
```

Obsérvese que este método no copia *v.vector* en *this.vector*, sino que realiza la copia de los elementos de *v.vector* en una nueva matriz apuntada *this.vector* del mismo tamaño. Si no hiciera esto, tendríamos una sola matriz referenciada por dos objetos.

- *destructor*: método que permite liberar el espacio de memoria ocupado por la matriz que encapsula un objeto *CVector*.

```
CVector::~~CVector()
{
    delete [] vector;
}
```

- *elemento*: método que devuelve el dato almacenado en el elemento especificado de un objeto *CVector*.

```
double& CVector::elemento(int i)
{
    return vector[i];
}
```

- *longitud*: método que devuelve el número de elementos de un objeto *CVector*.

```
int CVector::longitud() const
{
    return nElementos;
}
```

- *asignarMem*: método que devuelve un puntero al bloque de memoria necesario para construir la matriz que encapsula un objeto *CVector*.

```
double *CVector::asignarMem(int nElems)
{
    try
    {
        double *p = new double[nElems];
        return p;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        exit(-1);
    }
}
```

El resultado de encapsular los atributos y los métodos anteriormente expuestos es la clase *CVector* que se muestra a continuación:

```
// vector.h - Declaración de la clase CVector
#ifndef _VECTOR_H_
#define _VECTOR_H_

class CVector
{
private:
    double *vector; // puntero al primer elemento de la matriz
```

```
    int nElementos; // número de elementos de la matriz
protected:
    double *asignarMem(int);
public:
    CVector(); // crea un CVector con N de elementos por omisión
    CVector(int ne); // crea un CVector con ne elementos
    CVector(double *, int); // crea un CVector desde una matriz
    CVector(const CVector&); // crea un CVector desde otro
    ~CVector(); // destructor
    double& elemento(int i);
    int longitud() const;
};

#endif // _VECTOR_H_
```

```
// vector.cpp - Definición de la clase CVector
```

```
#include <iostream>
#include "vector.h"
using namespace std;

// Constructores:
// Crear una matriz con 10 elementos por omisión
CVector::CVector()
{
    // ...
}

// Crear una matriz con ne elementos
CVector::CVector(int ne)
{
    // ...
}

// Crear una matriz a partir de otra matriz primitiva
CVector::CVector(double *a, int ne)
{
    // ...
}

// Constructor copia
CVector::CVector(const CVector& v)
{
    // ...
}

// Otros métodos
CVector::~CVector() // destructor
{
    delete [] vector;
}
```

```

double& CVector::elemento(int i)
{
    return vector[i];
}
int CVector::longitud() const
{
    return nElementos;
}

double *CVector::asignarMem(int nElems)
{
    // ...
}

```

El resultado es que cada objeto *CVector* consta de dos bloques de memoria: uno de tamaño fijo que almacena su estructura interna (*vector* y *nElementos*) y otro de una longitud elegida que almacena los datos (la matriz de tipo **double**).

Para probar la clase expuesta escriba, por ejemplo, la siguiente aplicación:

```

// test.cpp - Miembros que son punteros
#include <iostream>
#include <iomanip>
#include "vector.h"
using namespace std;
void fnVectores();
void fnVisualizar(CVector&);

int main()
{
    CVector vector1, vector2(5);
    fnVisualizar(vector1);
    fnVisualizar(vector2);
    CVector vector3 = vector2;
    fnVisualizar(vector3);
    fnVectores();
    cout << "fin del programa\n";
}

void fnVectores()
{
    double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz primitiva
    CVector vector4(x, sizeof(x)/sizeof(double));
    fnVisualizar(vector4);
    CVector *pvector5 = new CVector(10);
    fnVisualizar(*pvector5);
    CVector vector6 = *pvector5;
    fnVisualizar(vector6);
    delete pvector5; // también: pvector5->CVector::~~CVector();
}

```

```

void fnVisualizar(CVector& vector)
{
    int ne = vector.longitud();
    for (int i = 0; i < ne; i++)
        cout << setw(7) << vector.elemento(i);
    cout << "\n\n";
}

```

Analizando a grandes rasgos el código presentado anteriormente, podemos ver que la definición:

```
CVector vector1;
```

llama al constructor *CVector* y crea un objeto *vector1* con 10 elementos por omisión. La definición:

```
CVector vector2(5);
```

llama al constructor *CVector(int ne)* y crea un objeto *vector2* con cinco elementos. También, cualquiera de las líneas:

```

CVector vector2 = CVector(5); // conversión explícita de 5 a CVector
CVector vector2 = 5;          // conversión implícita de 5 a CVector
CVector *pvector2 = new CVector(5); // conversión explícita

```

invocarían al constructor *CVector(int ne)* y crearían un vector de cinco elementos, excepto si el constructor se califica **explicit** (*explicit CVector(int ne)*). Un constructor calificado **explicit** sólo permite conversiones explícitas (véase en el capítulo siguiente *Conversión mediante constructores*). La línea:

```
CVector vector3 = vector2;
```

llama al constructor copia y crea un objeto *vector3* iniciado con los datos del objeto *vector2*. Las líneas:

```

double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz x
CVector vector4(x, sizeof(x)/sizeof(double));

```

definen, la primera, la matriz *x* y la última llama al constructor *CVector(double *, int)* que crea un objeto *vector4* iniciado con los datos de la matriz *x*.

Como se puede observar, cada vez que se crea un objeto es llamado automáticamente un constructor, lo que garantiza la iniciación del objeto. El que se llame a uno o a otro, depende del número y tipo de los argumentos especificados.

El destructor *~CVector* permite liberar la memoria asignada dinámicamente de una forma automática; esto es, cada vez que el flujo de ejecución sale del ámbito de un determinado objeto, el destructor es llamado automáticamente. En el ejemplo, el ámbito de los objetos *vector4*, **pvector5* y *vector6* está limitado a la función *fnVectores*. Cuando finalice la ejecución de esta función, el destructor

será llamado una vez por cada objeto, liberándose así la memoria que ocupan. Lo mismo ocurrirá con *vector1*, *vector2* y *vector3*, cuando finalice la función **main**.

Observe el objeto referenciado por *pvector5*. Ha sido creado utilizando el operador **new**, por lo que debe utilizarse explícitamente el operador **delete** para que el destructor sea llamado automáticamente; esto es:

```
delete pvector5;
```

También podíamos haber invocado al destructor explícitamente utilizando la siguiente sintaxis, aunque, como hemos visto, no hay necesidad de hacer esto:

```
pvector5->CVector::~~CVector();
```

La clase *CVector* es un ejemplo típico de una clase que requiere un destructor. ¿Por qué? Porque cuando el flujo de ejecución sale fuera del ámbito de un determinado objeto *CVector*, el bloque de memoria que almacena la estructura interna de ese objeto (el puntero *vector* a una matriz de tipo **double**, no la matriz, y el entero *nElementos*) se libera automáticamente. Sin embargo, la memoria asignada por **new** para la matriz referenciada por *vector* no se libera si no se indica explícitamente, razón por la que se necesita definir un destructor que utilizando el operador **delete** realice esta operación.

```
~CVector() { delete [] vector; }
```

Sin embargo, una clase con atributos que son punteros a otros objetos, como es *CVector*, potencialmente tiene problemas. Para comprobarlo, suponga que el constructor copia de la clase *CVector* se hubiera escrito así:

```
CVector::CVector(const CVector& v)
{
    nElementos = v.nElementos;
    vector = v.vector;
}
```

Suponga también que la función **main** de la aplicación anterior fuera como sigue:

```
int main()
{
    double x[] = { 1, 2, 3, 4, 5, 6, 7 }; // matriz x
    CVector vector1(x, 7);
    fnVisualizar(vector1); // escribe 1 2 3 4 5 6 7

    // El siguiente bloque define vector2
    {
```

```

CVector vector2 = vector1;
for (int i = 0; i < vector2.longitud(); i++)
    vector2.elemento(i) *= 10;
fnVisualizar(vector2); // escribe 10 20 30 40 50 60 70
}
// vector2 ha sido destruido
fnVisualizar(vector1); // escribe ?

cout << "Fin de la aplicación\n";
}

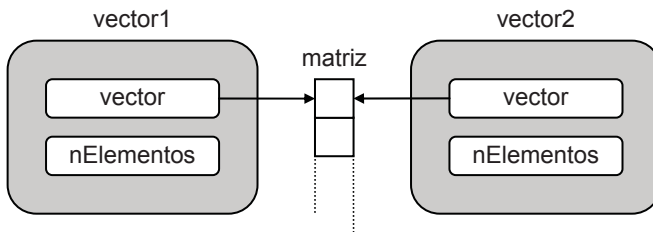
```

Ahora la función **main** crea un objeto *vector1* iniciado con los valores de una matriz *x* e incluye un bloque que crea un nuevo objeto *vector2* a partir de *vector1*, para lo cual se invoca al constructor copia.

Observe que ahora este constructor simplemente copia los atributos del objeto *v* en los correspondientes atributos del nuevo objeto creado. Por lo tanto, el resultado de una sentencia como:

```
CVector vector2 = vector1;
```

será dos objetos, *vector1* y *vector2*, referenciando la misma matriz. La figura siguiente muestra esto con claridad:



Esto significa que cualquier modificación en uno de los objetos afectará a ambos, justo lo que sucede cuando se ejecuta el código siguiente. Las modificaciones realizadas en el objeto *vector2* afectan de la misma forma a *vector1*:

```

// El siguiente bloque define vector2
{
    CVector vector2 = vector1;
    for (int i = 0; i < vector2.longitud(); i++)
        vector2.elemento(i) *= 10;
    fnVisualizar(vector2); // escribe 10 20 30 40 50 60 70
}

```

Piense ahora qué sucederá cuando el flujo de ejecución salga fuera del ámbito de *vector2*. Pues que el objeto *vector2* será eliminado, lo que implica ejecutar el destructor de la clase *CVector*, que liberará el bloque de memoria correspondiente

a la matriz. Esto significa que cualquier operación posterior con el objeto *vector1* puede dar lugar a resultados inesperados, ya que el atributo *vector* de *vector1* apuntaba al mismo bloque de memoria y éste ha sido liberado.

Además, cuando el flujo de ejecución salga fuera del ámbito de *vector1* se invocará otra vez al destructor, que intentará liberar de nuevo el bloque de memoria de la matriz de tipo **double**, lo que puede ocasionar resultados inesperados.

Esta misma teoría es aplicable al método **operator=**. Esto significa que debemos poner un especial interés cuando escribamos métodos que tengan como finalidad duplicar objetos que tienen atributos que son punteros a otros objetos (véase *Asignación* en el capítulo siguiente: *Operadores sobrecargados*).

En el capítulo *Tipos estructurados de datos* vimos que la biblioteca estándar de C++ proporciona un contenedor **vector** a partir del cual podemos trabajar con matrices dinámicas de cualquier tipo y dimensión. Nuestra clase *CVector* no es más que una introducción a lo que puede ser un contenedor de este tipo.

MIEMBROS STATIC DE UNA CLASE

En ocasiones se hace necesario disponer de una variable global públicamente accesible que no sea parte de un objeto, pero sí de la clase, o bien de un método que no necesite ser invocado para un objeto en particular. Esta funcionalidad es proporcionada en C++ por los miembros estáticos (**static**) de una clase.

Atributos static

La última versión de la clase *CFecha* definía un constructor que asignaba una fecha por omisión si la pasada como argumento no era correcta. No obstante, sería más conveniente disponer de un valor por omisión, compartido por todos los objetos de la clase, que además pudiera ser modificado por el usuario, pudiendo incluso llegar a tomar el valor de la fecha actual. Esto se traduce en un atributo del cual sólo es necesario que exista una única copia que pueda ser utilizada por todos los objetos *CFecha*; esto es, una variable con ámbito global accesible directamente, o bien indirectamente a través de un método de la interfaz. La alternativa que ofrece C++ para dar solución al problema planteado es declarar el atributo **static**.

Un atributo **static** no es un atributo específico de un objeto (el *día* sí es un atributo específico de una fecha; cada fecha tiene su día), sino más bien es un atributo de la clase; esto es, un atributo del que sólo hay una copia que comparten todos los objetos de la clase. Por esta razón, un atributo **static** existe y puede ser utilizado aunque no exista ningún objeto de la clase.

Como ejemplo, vamos a asociar un atributo *fechaPorOmission* con la clase, no con cada objeto. El código mostrado a continuación muestra cómo hacerlo:

```
class CFecha
{
    // Atributos
    private:
        int dia, mes, anyo;
        static CFecha fechaPorOmission;
    // ...
};
```

Un atributo **static** puede ser calificado como **private** (privado), **protected** (protegido) o **public** (público). También, podemos calificarlo **const** para que sea una constante en lugar de una variable. Así mismo, tiene que ser iniciado a nivel global (ámbito de fichero, no de clase), porque la declaración de un atributo en su clase no se considera una definición, de ahí que haya que iniciarlos explícita o implícitamente.

La iniciación de un atributo **static** se coloca generalmente en el fichero fuente *.cpp* que contiene las definiciones de los métodos de la clase. No se puede realizar la iniciación en un lugar donde se pueda producir más de una vez; por ejemplo, en un fichero de cabecera, ya que éste puede cargarse desde varios ficheros *.cpp* del mismo proyecto, lo que daría lugar a otras tantas iniciaciones, produciéndose durante el enlace un error por redefinición.

Por ejemplo, podríamos iniciar el atributo *fechaPorOmission* en el fichero *fecha.cpp* como se indica a continuación, y utilizarlo en el constructor de la clase para iniciar un objeto cuando la fecha pasada como argumento sea incorrecta:

```
// fecha.cpp - Definición de los métodos de la clase CFecha
// ...
CFecha CFecha::fechaPorOmission = CFecha(1, 1, 2001);

CFecha::CFecha(int dd, int mm, int aaaa) : /* constructor */
    dia(dd), mes(mm), anyo(aaaa)
{
    if (!fechaCorrecta())
    {
        cout << "Fecha incorrecta. Se asigna la fecha por omisión.\n";
        *this = fechaPorOmission;
    }
}
// ...
```

Obsérvese que iniciar un atributo estático supone definirlo a nivel global, fuera de la clase, que no hay que especificar de nuevo la palabra **static** y, lógicamente,

te, sí hay que especificar la clase a la que pertenece. Hay una excepción: cuando el atributo se declara entero, **static** y **const**, sí puede iniciarse en la declaración de la clase. Veamos otros ejemplos:

```
// fichero .h
class X
{
    static char a[10][80];
    static int b;
    static std::string s;
    static const int K = 10;
};

// fichero .cpp
char X::a[10][80];
int X::b = 0;
std::string X::s = std::string("error");
```

Acceder a los atributos static

En el apartado anterior podemos ver cómo los métodos de la clase *CFecha*, por ejemplo el constructor, pueden acceder directamente al atributo *fechaPorOmission* de la misma, igual que acceden al resto de los atributos. Pero, desde otra clase o desde cualquier función externa, ¿cómo podríamos acceder a esa información? Si *fechaPorOmission* fuera un atributo declarado **public**, podríamos acceder a él directamente a través del nombre de la clase (utilizar el nombre de un objeto, aunque es válido, puede dar lugar a malas interpretaciones del código). Por ejemplo:

```
int main()
{
    CFecha fecha1(1, 3, 2012);
    if (fecha1.fechaCorrecta())
        visualizarFecha(fecha1);
    visualizarFecha(CFecha::fechaPorOmission);
}
```

También se podría haber escrito *fecha1.fechaPorOmission*, no obstante no se aconseja porque da idea de que ese atributo pertenece al objeto *fecha1*, lo cual es falso. Se puede observar que *CFecha::fechaPorOmission* se comporta como si se tratara de una variable pública, ya que utilizando esta sintaxis podemos acceder a *fechaPorOmission* desde cualquier otra clase o función externa.

Puesto que el atributo *fechaPorOmission* se ha declarado privado y no público, tendremos que añadir a la interfaz de la clase un método público que permita el acceso al mismo y, preferiblemente, que se comporte como el atributo.

Métodos **static**

Un método declarado **static** carece del puntero **this** por lo que no puede ser invocado para un objeto de su clase, sino que se invoca en general allí donde se necesite utilizar la operación para la que ha sido escrito. Desde este punto de vista es imposible que un método **static** pueda acceder a un miembro no **static** de su clase; por la misma razón, sí puede acceder a un miembro **static**.

Como ejemplo, vamos a añadir a la clase *CFecha* un método *asignarFechaPorOmission* estático que permita cambiar la fecha de *fechaPorOmission*.

```
void CFecha::asignarFechaPorOmission(int dd, int mm, int aaaa)
{
    fechaPorOmission.asignarFecha(dd, mm, aaaa);
}
```

Un método se declara **static** en el cuerpo de la clase, no en la definición externa a la clase, y sólo puede acceder a los miembros (atributos o métodos) **static** de su clase. Por ejemplo, si en el método anterior intenta establecer el atributo *día* a 1, el compilador le mostrará un error indicándole que no se puede hacer referencia a un atributo no estático desde un método estático.

En cambio, un miembro **static** sí puede ser accedido por un método independientemente de que sea **static** o no. Por ejemplo, antes hemos visto que el constructor *CFecha* accede al atributo **static** *fechaPorOmission*.

Si el acceso al método **static** se hace desde un método de otra clase o desde una función externa, dicho método tiene que ser invocado a través del nombre de la clase según se explicó anteriormente para los atributos **static**. Por ejemplo:

```
int main()
{
    CFecha::asignarFechaPorOmission(); // sin parámetros, fecha actual
    CFecha fecha1(29, 2, 2005);
    visualizarFecha(fecha1);
}
```

Se puede observar que el comportamiento de *CFecha::asignarFechaPorOmission* es igual que el de cualquier otro método de un lenguaje no orientado a objetos. Esto hace posible escribir programas C++ utilizando solamente esta clase de métodos, pero entonces se frustraría el propósito más importante de este lenguaje: la POO. No piense por ello que utilizar este tipo de métodos es una trampa. Hay muchas y buenas razones para utilizarlos; y si no, observe la utilidad del método siguiente. Se trata de un método **static** de la clase *CFecha* que obtiene la fecha actual del sistema:

```

void CFecha::obtenerFechaActual(int& dd, int& mm, int& aaaa)
{
    // Obtener la fecha actual.
    struct tm *fh;
    time_t segundos;
    time(&segundos);
    fh = localtime(&segundos);
    dd = fh->tm_mday; mm = fh->tm_mon+1; aaaa = fh->tm_year+1900;
}

```

Ahora, este método puede utilizarlo indistintamente en los métodos de la clase *CFecha*, en los de cualquier otra clase o en las funciones externas. Por ejemplo, vamos a modificar el método *asignarFecha* de la clase *CFecha* así:

```

void CFecha::asignarFecha(int dd, int mm, int aaaa)
{
    int d, m, a;
    obtenerFechaActual(d, m, a);
    // Asignar por omisión la fecha actual.
    if (dd) dia = dd; else dia = d;
    if (mm) mes = mm; else mes = m;
    if (aaaa) anyo = aaaa; else anyo = a;
}

```

Después de todos estos cambios realizados, la declaración de la clase *CFecha* queda así:

```

class CFecha
{
    // Atributos
    private:
        int dia, mes, anyo;
        static CFecha fechaPorOmision; // se define en fecha.cpp
    // Métodos
    protected:
        bool bisiestro() const;
    public:
        CFecha(int dd = 1, int mm = 1, int aaaa = 2001); // constructor
        void asignarFecha(int dd = 0, int mm = 0, int aaaa = 0);
        void obtenerFecha(int& dd, int& mm, int& aaaa) const;
        bool fechaCorrecta() const;
        const int& obtenerDia() const;
        const int& obtenerMes() const;
        const int& obtenerAnyo() const;
        static void asignarFechaPorOmision(int = 0, int = 0, int = 0);
        static void obtenerFechaActual(int&, int&, int&);
};

```

ATRIBUTOS QUE SON OBJETOS

Un miembro de una clase puede ser un objeto **static** de su misma clase (un ejemplo es la clase *CFecha* del apartado anterior), un objeto de otra clase (inclusión) o bien un puntero a un objeto de su misma u otra clase (delegación). Por ejemplo, pensemos en los cumpleaños de aquellas personas a las que deseamos felicitar con la intención de escribir una clase de objetos que represente estos eventos. *Nombre*, *fecha de nacimiento* y *teléfono* son atributos básicos para este tipo de objetos, donde *fecha de nacimiento* puede ser un atributo de la clase *CFecha* a la que nos hemos referido anteriormente. Como ejemplo, vamos a agrupar estos atributos y los métodos para manipularlos en una clase *CCumpleanyos* como la siguiente:

```
class CCumpleanyos
{
private:
    char *nombre;
    CFecha fecha_nacimiento; // objeto CFecha
    long telefono;
protected:
    char *asignarCadena(char *);
public:
    CCumpleanyos(char *n = 0, int d = 1, int m = 1, int a = 1997);
    ~CCumpleanyos() { delete [] nombre; }
    void asignarNombre(char *);
    char *obtenerNombre(char *);
    CFecha& fechaNacimiento() { return fecha_nacimiento; }
    void asignarTelefono(long);
    long obtenerTelefono();
};
```

Esta declaración especifica que el atributo privado *fecha_nacimiento* es un objeto de la clase *CFecha*. Este diseño indica que cada operación de *CFecha* no definida en la clase *CCumpleanyos* puede ser servida por el objeto *fecha_nacimiento*, lo que se denomina inclusión mediante objetos (otros estudiosos del tema lo denominan agregación de objetos). Por ejemplo, si quisiéramos añadir un método para asignar la fecha de cumpleaños, esta operación podría ser servida por el método *asignarFecha* de *CFecha*:

```
void CCumpleanyos::asignarFechaCumple(int dd, int mm, int aaaa)
{
    fecha_nacimiento.asignarFecha(dd, mm, aaaa);
}
```

La siguiente línea crea un objeto de la clase *CCumpleanyos*:

```
CCumpleanyos persona( "Francisco", 25, 8, 1982, 666777888 );
```

Cuando se crea un objeto *CCumpleanyos*, primero se invoca al constructor de *CCumpleanyos*, que a su vez invoca al constructor de *CFecha*, se ejecuta *CFecha*, después *CCumpleanyos* y finaliza así la construcción del objeto. En general, para llamar al constructor de un atributo, se aconseja utilizar una lista de iniciadores; esto es, coloque dos puntos a continuación de la lista de parámetros del constructor de su clase y luego especifique el nombre del atributo seguido de los argumentos encerrados entre paréntesis, cuestión que ya vimos anteriormente en este capítulo. Por ejemplo, el constructor *CCumpleanyos* puede ser el siguiente:

```
// Declaración del constructor CCumpleanyos
CCumpleanyos(char * = 0,int = 1,int = 1,int = 2001,long = 0);

// ...

// Definición del constructor CCumpleanyos
CCumpleanyos::CCumpleanyos(char *nom,int dd,int mm,int aa,long tel):
fecha_nacimiento(dd, mm, aa), telefono(tel)
{
    nombre = asignarCadena(nom);
}
```

La sintaxis empleada en la definición del cuerpo del constructor *CCumpleanyos* indica que antes de que comience a ejecutarse el cuerpo del mismo, hay que invocar al constructor *CFecha* para iniciar el atributo *fecha_nacimiento* con los argumentos especificados.

Si la clase tuviera más de un atributo que iniciar, como sucede en el ejemplo, entonces especifique la lista de iniciadores correspondiente separados por comas.

Si en la definición del constructor *CCumpleanyos* no se especifica el iniciador para el atributo *fecha_nacimiento*, el compilador ejecutará primero el constructor por omisión de la clase *CFecha* y después el constructor *CCumpleanyos*. En este caso, la iniciación del objeto *CFecha* puede hacerse a través de sus métodos de acceso. Por ejemplo:

```
CCumpleanyos::CCumpleanyos(char *nom,int dd,int mm,int aa,long tel):
telefono(tel)
{
    nombre = asignarCadena(nom);
    fecha_nacimiento.asignarFecha(dd, mm, aa);
}
```

Sin embargo, esta forma de operar no es eficiente, ya que el objeto *fecha_nacimiento* es iniciado dos veces, una por el constructor por omisión y otra por el método *asignarFecha*. En general, es aconsejable especificar la lista de iniciadores para iniciar los atributos de la clase.

También podríamos haber diseñado la clase *CCumpleanyos* para que el atributo *fecha_nacimiento* fuera un puntero a un objeto *CFecha*:

```
class CCumpleanyos
{
private:
    char *nombre;
    CFecha *fecha_nacimiento; // puntero a un objeto CFecha
    // ...
};
```

Este diseño especifica que cada operación de *CFecha* no definida en la clase *CCumpleanyos* puede ser servida por el puntero *fecha_nacimiento*, lo que se denomina delegación mediante objetos. Por ejemplo, en este caso, el método para asignar la fecha de cumpleaños podríamos escribirlo así:

```
void CCumpleanyos::asignarFechaCumple(int dd, int mm, int aaaa)
{
    fecha_nacimiento->asignarFecha(dd, mm, aaaa);
}
```

CLASES INTERNAS

Una *clase interna* es una clase que es un miembro de otra clase. Por ejemplo, en el código mostrado a continuación, *Punto* es una clase interna:

```
// circulo.h - Declaración de la clase Circulo
#ifndef _CIRCULO_H_
#define _CIRCULO_H_

class Circulo
{
private:
    class Punto
    {
private:
        double x, y;

public:
        Punto(double cx = 0, double cy = 0) { x = cx; y = cy; }
        double X() const { return x; }
        double Y() const { return y; }
    };
    Punto centro; // coordenadas del centro
    double radio; // radio del círculo

protected:
    void msgEsNegativo() const;
```



```
public:
    Circulo() {} // constructor sin parámetros
    Circulo(double cx, double cy, double r); // constructor
    double longCircunferencia() const;
    double areaCirculo() const;
    void coordenadasCentro(double& x, double& y) const;
};

#endif // _CIRCULO_H_

// circulo.cpp - Definición de los métodos de la clase Circulo
#include <iostream>
#include "circulo.h"
using namespace std;

void Circulo::msgEsNegativo() const
{
    cout << "El radio es negativo. Se convierte a positivo\n";
}

Circulo::Circulo(double cx, double cy, double r) : centro(cx, cy)
{
    if (r < 0)
    {
        msgEsNegativo();
        r = -r;
    }
    radio = r;
}

double Circulo::longCircunferencia() const
{
    return 2 * 3.1415926 * radio;
}

double Circulo::areaCirculo() const
{
    return 3.1415926 * radio * radio;
}

void Circulo::coordenadasCentro(double& x, double& y) const
{
    x = centro.X();
    y = centro.Y();
}

// test.cpp - Clases internas
#include <iostream>
#include "circulo.h"
using namespace std;
```

```

int main()
{
    Circulo c(100, 120, 10);
    cout << c.areaCirculo() << endl;
    double x, y;
    c.coordenadasCentro(x, y);
    cout << "x = " << x << ", y = " << y << endl;
}

```

Una clase se debe definir dentro de otra sólo cuando tenga sentido en el contexto de la clase que la incluye o cuando depende de la función que desempeña la clase que la incluye. Por ejemplo, una ventana puede definir su propio cursor; en este caso, la ventana puede ser un objeto de una clase y el cursor de una clase anidada de ésta. También, una clase puede definir sus propias excepciones, como veremos más adelante en otro capítulo.

Una clase interna es un miembro más de la clase que la contiene. En el ejemplo anterior la clase *Punto* es un miembro más de *Circulo* y como tal se le aplican las mismas reglas que para el resto de los miembros. Según esto, *Punto* tendrá acceso al resto de los miembros de *Circulo* independientemente de su modificador de acceso (decir *Punto* implica a los miembros de *Punto*); *Punto* puede ser pública, privada o protegida.

INTEGRIDAD DE LOS DATOS

Parece que proporcionar capacidades tanto de asignar como de obtener resulta, en esencia, lo mismo que declarar públicos los atributos privados. Pero no es así, ya que si un atributo se declara público puede ser escrito o leído a voluntad por cualquier función del programa. En cambio, si un atributo es privado, son los métodos los que permiten a otros métodos o funciones externas escribir o leer su contenido, pero controlando el formato y tratamiento del atributo. Esto garantiza la integridad de los datos.

Por ejemplo, en la clase *CCumpleanyos* que vimos anteriormente, para acceder al contenido del objeto *fecha_nacimiento*, se ha definido el método *fechaNacimiento*, que devuelve una referencia a un objeto *CFecha*, lo que permite utilizar dicho método a ambos lados del operador de asignación: a la izquierda cuando necesitemos asignar una fecha y a la derecha cuando necesitemos obtenerla. Entonces, son los métodos de la clase *CFecha* los que tienen que garantizar la integridad de los datos de un objeto *CFecha*.

```
CFecha& fechaNacimiento() { return fecha_nacimiento; }
```

Otro ejemplo; para acceder al contenido del atributo *nombre*, se ha definido el método de acceso *asignarNombre*, que a través del método *asignarCadena* garantiza que siempre se asigne una cadena a *nombre*, a menos que el puntero pasado sea nulo, y *obtenerNombre*, que para retornar *nombre* lo copia primero en una nueva cadena pasada como argumento, salvaguardando así su integridad.

```
void CCumpleanyos::asignarNombre(char *nom)
{
    delete [] nombre;
    nombre = asignarCadena(nom);
}

char *CCumpleanyos::obtenerNombre(char *nom)
{
    if (nom && nombre)
        strcpy(nom, nombre);
    return nom;
}
```

Observe que *asignarNombre* en primer lugar libera la memoria que pudiera tener asignada *nombre* de una operación anterior.

El hecho de que *obtenerNombre* devuelva la cadena copiada en *nom* permite utilizar dicho método de dos formas: una, para asignar *nombre* a una nueva cadena pasada por referencia y dos, en expresiones:

```
char nom[80];
// Primer caso: asignar el atributo nombre a nom
persona.ObtenerNombre(nom);

// Segundo caso: mostrar el valor devuelto por obtenerNombre; este
// valor también ha sido copiado en nom.
cout << persona.ObtenerNombre(nom) << endl;
```

Resumiendo, controlar el acceso a los atributos privados, especialmente en los intentos de asignar un valor, garantiza la integridad de los datos.

El siguiente programa implementa una mínima función **main** para probar la funcionalidad de la clase *CCumpleanyos*.

```
// test.cpp - cumple años
#include <iostream>
#include "fecha.h"
#include "cumple.h"

using namespace std;
```

```
void visualizar(CCumpleanyos&);

int main()
{
    char nombre[80];
    int dia, mes, anyo;
    long telefono;

    CCumpleanyos p1("Francisco", 25, 8, 1982, 666777888);
    visualizar( p1 );

    CCumpleanyos p2;

    cout << "Nombre:     ";
    cin.getline ( nombre, 80, '\n' );
    p2.asignarNombre( nombre );

    cout << "Fecha de nacimiento\n";
    cout << "día: "; cin >> dia;
    cout << "mes: "; cin >> mes;
    cout << "año: "; cin >> anyo;
    CFecha fecha(dia, mes, anyo);
    p2.fechaNacimiento() = fecha;

    cout << "Teléfono: ";
    cin >> telefono;
    p2.asignarTelefono(telefono);

    visualizar(p2);
}

// Visualizar un objeto
void visualizar(CCumpleanyos& persona)
{
    int dia, mes, anyo;
    char nombre[80];
    persona.fechaNacimiento().obtenerFecha(dia, mes, anyo);
    cout << persona.obtenerNombre(nombre) << endl
         << dia << "/" << mes << "/" << anyo << endl
         << persona.obtenerTelefono() << endl;
}
```

DEVOLVER UN PUNTERO O UNA REFERENCIA

Un método público que devuelva un puntero a un atributo privado vulnera la encapsulación de la clase, a no ser que el puntero sea a un elemento de datos constante. Por ejemplo, suponga que hubiéramos escrito el método *obtenerNombre* así:

```
char *CCumpleaños::obtenerNombre() { return nombre; }
```

Observe que el método *obtenerNombre* devuelve un puntero a una cadena de caracteres, concretamente a *nombre*, lo que significa que *obtenerNombre* permite obtener la dirección *nombre*, dirección de comienzo de la cadena de caracteres. Entonces, unas líneas de código como las siguientes modificarían la cadena referenciada por *nombre* violando la característica de ocultación de datos:

```
char *pnombre = p2.obtenerNombre(); // pnombre = nombre
strcpy(pnombre, "hola"); // nombre apunta a "hola"
```

Podríamos optar por retornar un puntero a un objeto constante (en este caso cadena de caracteres), impidiendo así la modificación del objeto devuelto:

```
const char* CCumpleaños::obtenerNombre() { return nombre; }
```

pero esta forma de proceder tampoco asegura nada, ya una simple conversión explícita de **const char *** a **char *** vulneraría la seguridad.

En el ejemplo anterior, trabajar con *pnombre* es trabajar con el atributo *nombre*, y con *fechaNacimiento*, que devolvía una referencia al dato miembro *fecha_nacimiento*, ocurre algo similar según demuestra el ejemplo siguiente:

```
CFecha& fn = p2.fechaNacimiento(); // fn = fecha_nacimiento
fn.asignarFecha(1,1,2020);
```

Sin embargo, la integridad del dato *fecha_nacimiento* del objeto *CFecha* está salvada en la medida de la seguridad que ofrezcan los métodos de su interfaz pública, cosa que no ocurría con *nombre* de *CCumpleaños*.

MATRICES DE OBJETOS

Se puede crear una matriz de objetos de cualquier clase, de la misma forma que se crea una matriz de números, de caracteres, de objetos **string**, etc. Por ejemplo, suponiendo que tenemos definida una clase *CPersona*, podemos definir la matriz *listaTelefonos* con 100 elementos de la forma siguiente:

```
CPersona listaTelefonos[100];
```

listaTelefonos es una matriz de objetos de la clase *CPersona*. Cada elemento de esta matriz será iniciado por un constructor *CPersona* sin argumentos. Por lo tanto, si la clase no tiene definido un constructor, se utilizará el constructor por omisión que iniciará cada elemento de la matriz con los valores predeterminados por el sistema según la matriz sea local o global. Si la clase tiene definido un

constructor, que es lo normal, dicho constructor no debe tener argumentos o tenerlos con valores por omisión; de lo contrario, una declaración como la anterior daría lugar a un error.

Como ejemplo, supongamos que deseamos mantener una lista de teléfonos. La lista será un objeto que encapsule la matriz de objetos persona y que muestre una interfaz que permita añadir, eliminar y buscar una en la lista.

En un primer análisis sobre el enunciado identificamos dos clases de objetos: personas y lista de teléfonos.

La clase de objetos persona (que denominaremos *CPersona*) encapsulará el nombre, la dirección y el teléfono de cada una de las personas de la lista; así mismo, proporcionará la funcionalidad necesaria para establecer u obtener los datos de cada persona individual.

El listado siguiente muestra un ejemplo de una clase *CPersona* que define los atributos privados *nombre*, *dirección* y *teléfono* relativos a una persona y los métodos públicos que forman la interfaz de esta clase de objetos:

- Constructores, con y sin argumentos, para iniciar un objeto persona.
- Métodos de acceso (*asignar...* y *obtener...*) para cada uno de los atributos.

```
// persona.h - Declaración de la clase CPersona
#ifndef _PERSONA_H_
#define _PERSONA_H_
#include <string>
using namespace std;

class CPersona
{
private:
    string nombre;
    string direccion;
    long telefono;
public:
    CPersona(string nom = "", string dir = "", long tel = 0);
    void asignarNombre(string nom);
    string obtenerNombre() const;
    void asignarDireccion(string dir);
    string obtenerDireccion() const;
    void asignarTelefono(long tel);
    long obtenerTelefono() const;
};

#endif // _PERSONA_H_
```

```
// persona.cpp - Definición de los métodos de la clase CPersona
#include <iostream>
#include "persona.h"

CPersona::CPersona(string nom, string dir, long tel):
nombre(nom), direccion(dir), telefono(tel)
{
}

void CPersona::asignarNombre(string nom)
{
    nombre = nom;
}

string CPersona::obtenerNombre() const
{
    return nombre;
}

void CPersona::asignarDireccion(string dir)
{
    direccion = dir;
}

string CPersona::obtenerDireccion() const
{
    return direccion;
}

void CPersona::asignarTelefono(long tel)
{
    telefono = tel;
}

long CPersona::obtenerTelefono() const
{
    return telefono;
}
```

Un método como *asignarNombre* simplemente asigna el nombre pasado como argumento al atributo *nombre* del objeto que recibe el mensaje. Y un método como *obtenerNombre* devuelve el atributo *nombre* del objeto que recibe el mensaje. La explicación para los otros métodos es análoga.

Por ejemplo, el código siguiente crea una matriz primitiva *listaTfnos* de *N* objetos *CPersona*, asigna al elemento *listaTfnos[i]* un nombre y posteriormente lo muestra:

```

int main()
{
    const int N = 100;
    CPersona listaTfnos[N];
    int i = 0;
    // ...
    listaTfnos[i].asignarNombre("Javier");
    cout << listaTfnos[i].obtenerNombre() << endl; // escribe: Javier
}

```

Este otro ejemplo mostrado a continuación crea una matriz *listaTfnos* de *N* punteros a objetos *CPersona*, la inicia a cero, asigna al elemento *listaTfnos[i]* un nombre, lo muestra y finalmente libera la memoria asignada para cada objeto:

```

int main()
{
    const int N = 100;
    CPersona *listaTfnos[N]; // matriz de punteros a objetos CPersona
    fill(listaTfnos, listaTfnos + N, static_cast<CPersona *>(0));
    int i = 0;
    // ...
    listaTfnos[i] = new CPersona; // crear un objeto CPersona
    listaTfnos[i]->asignarNombre("Javier");
    cout << listaTfnos[i]->obtenerNombre() << endl;
    // Liberar la memoria asignada para cada objeto
    for (int i = 0; i < N; i++)
        if (listaTfnos[i]) delete listaTfnos[i];
}

```

El siguiente ejemplo crea una matriz dinámica *listaTfnos* de *N* objetos *CPersona*, asigna al elemento *listaTfnos[i]* un nombre, lo muestra y finalmente libera la memoria asignada a la matriz:

```

int main()
{
    const int N = 100;
    CPersona *listaTfnos = new CPersona[N];
    int i = 0;
    // ...
    listaTfnos[i].asignarNombre("Javier");
    cout << listaTfnos[i].obtenerNombre() << endl; // escribe: Javier
    delete [] listaTfnos;
}

```

Este otro ejemplo mostrado a continuación, utilizando la plantilla **vector** de la biblioteca de C++, crea una matriz dinámica *listaTfnos* de *N* objetos *CPersona*, reserva un espacio inicial para 100 objetos (no crea los objetos), crea un objeto *CPersona*, le asigna un nombre y lo añade a la matriz, y finalmente lo muestra:


```

int main()
{
    const int N = 100;
    vector<CPersona> listaTfnos;
    listaTfnos.reserve(N); // reservar espacio para N elementos
    int i = 0;
    CPersona unaPersona;

    // ...

    unaPersona.asignarNombre("Javier");
    listaTfnos.push_back(unaPersona); // añadir un objeto a la matriz
    cout << listaTfnos[i].obtenerNombre() << endl; // escribe: Javier
}

```

El ejemplo anterior nos enseña que utilizando la biblioteca de C++ podemos, por ejemplo, manipular cadenas de caracteres (clase **string**) y trabajar con matrices dinámicas (plantilla **vector**) sin tener que gestionar nosotros la memoria, evitando así el trabajo con punteros y los errores que esto ocasiona (acceso a direcciones no válidas de memoria, lagunas de memoria, punteros nulos, etc.).

El listado siguiente muestra un ejemplo de lo que puede ser la clase lista de teléfonos, que denominaremos *CListaTfnos*. Define el atributo privado *listaTelefonos*, matriz de objetos *CPersona*, y los métodos que se describen a continuación:

```

class CListaTfnos
{
private:
    std::vector<CPersona> listaTelefonos; // matriz de objetos vacía

public:
    CListaTfnos(); // constructor
    CPersona registro(unsigned int i); // acceso al registro i
    void anyadir(CPersona obj); // añadir un registro al final
    bool eliminar(long tel); // eliminar un registro
    int buscar(string str, int pos); // buscar un registro
    size_t longitud();
};

```

Para crear un objeto lista de teléfonos escribiremos una línea de código como la siguiente:

```
CListaTfnos listatfnos;
```

Según este ejemplo, la clase *CListaTfnos* tiene que tener un constructor sin argumentos. ¿Qué puede hacer este constructor? El constructor *CListaTfnos* lo que puede hacer es reservar memoria para un número determinado de elementos.

La reserva de memoria con antelación tiene la ventaja de garantizar durante la ejecución de la aplicación, como mínimo, ese espacio de memoria.

```

CListaTfnos::CListaTfnos() // constructor
{
    // Reservar espacio para 100 elementos
    listaTelefonos.reserve(100);
}

```

Antes de que se ejecute el cuerpo del constructor anterior, el número de elementos de *listaTelefonos* vale 0 (resultado de haberse ejecutado el constructor de **vector**) y después de que se ejecute, sigue valiendo 0, aunque se ha adquirido memoria para almacenar los 100 objetos *CPersona* primeros. A partir de aquí, cada nuevo elemento que se añada requerirá también de una operación de adquisición de memoria.

Para añadir un teléfono (objeto *CPersona*) a la lista de teléfonos (objeto *CListaTfnos*) escribiremos un código análogo al siguiente:

```

listatfnos.anyadir(CPersona(nombre, direccion, telefono));

```

Cuando el objeto *listatfnos* de la clase *CListaTfnos* recibe el mensaje *anyadir*, responde ejecutando su método *anyadir* que añade al final de *listaTelefonos* el objeto *CPersona* pasado como argumento. Para realizar esta tarea, *anyadir* invoca al método **push_back** de **vector**.

```

void CListaTfnos::anyadir(CPersona objeto)
{
    listaTelefonos.push_back(objeto);
}

```

Para eliminar un teléfono (objeto *CPersona*) de la lista de teléfonos (objeto *CListaTfnos*) escribiremos un código análogo al siguiente:

```

eliminado = listatfnos.eliminar(telefono);

```

Cuando el objeto *listatfnos* de la clase *CListaTfnos* recibe el mensaje *eliminar*, responde ejecutando su método *eliminar* que quitará de la lista el elemento correspondiente al teléfono pasado como argumento y decrementará en 1 el tamaño de la lista. Para realizar estas dos tareas, primero buscará en la matriz *listaTelefonos* el objeto *CPersona* que tiene el número de teléfono pasado como argumento y, después, invocará al método **erase** para quitar ese elemento de la lista. El método *eliminar* devuelve **true** si se encontró y eliminó el elemento especificado y **false** en caso contrario.

```

bool CListaTfnos::eliminar(long tel)
{
    // Buscar el teléfono y eliminar registro
    for (unsigned int i = 0; i < listaTelefonos.size(); i++)
        if (listaTelefonos[i].obtenerTelefono() == tel)
        {
            listaTelefonos.erase(listaTelefonos.begin()+i);
            return true;
        }
    return false;
}

```

Para buscar un teléfono (objeto *CPersona*) en la lista de teléfonos (objeto *CListaTfnos*) escribiremos un código análogo al siguiente:

```
pos = listatfnos.buscar(cadenabuscar, posicion_inicio_búsqueda);
```

Cuando el objeto *listatfnos* de la clase *CListaTfnos* recibe el mensaje *buscar*, responde ejecutando su método *buscar* que recorrerá la lista de teléfonos en busca de un elemento (objeto *CPersona*) que contenga en su campo *nombre* la subcadena pasada como argumento. La búsqueda se iniciará en la posición pasada como argumento. El método *buscar* devolverá la posición del elemento buscado, si se encuentra, o -1 en caso contrario.

```

int CListaTfnos::buscar(string str, int pos)
{
    string nom;
    if (str.empty()) return -1;
    if (pos < 0) pos = 0;
    for (unsigned int i = pos; i < listaTelefonos.size(); i++)
    {
        nom = listaTelefonos[i].obtenerNombre();
        if (nom.empty()) continue;
        // ¿str está contenida en nom?
        if (nom.find(str) != string::npos)
            return i;
    }
    return -1;
}

```

Otros métodos de interés son *registro* y *longitud*. El método *registro* devuelve el objeto *CPersona* que está en la posición *i* de la matriz *listaTelefonos*, o un objeto con valores nulos si la posición especificada está fuera de límites.

```

CPersona CListaTfnos::registro(unsigned int i)
{
    if (i >= 0 && i < listaTelefonos.size())
        return listaTelefonos[i];
}

```

```
else
{
    cerr << "Índice fuera de límites\n";
    return CPersona();
}
}
```

El método *longitud* devuelve el número de elementos que tiene actualmente la matriz *listaTelefonos*.

```
size_t CListaTfnos::longitud()
{
    return listaTelefonos.size();
}
```

Hasta aquí, el diseño de las clases *CPersona* y *CListaTfnos*. El siguiente paso será escribir una aplicación que se ejecute así:

1. *Buscar*
2. *Buscar siguiente*
3. *Añadir*
4. *Eliminar*
5. *Salir*

```
Opción: 3
nombre:   Javier
dirección: Santander
teléfono: 942232323
```

1. *Buscar*
2. *Buscar siguiente*
3. *Añadir*
4. *Eliminar*
5. *Salir*

```
Opción:
```

A la vista del resultado anterior, esta aplicación mostrará un menú que presentará las operaciones que se pueden realizar sobre la lista de teléfonos. Posteriormente, la operación elegida será identificada por una sentencia **switch** y procesada de acuerdo al esquema presentado a continuación:

```
int menu()
{
    cout << "\n\n";
    cout << "1. Buscar\n";
    cout << "2. Buscar siguiente\n";
    cout << "3. Añadir\n";
}
```

```

    cout << "4. Eliminar\n";
    cout << "5. Salir\n";
    cout << endl;
    cout << "  Opción: ";
    int op;
    do
        cin >> op;
    while (op < 1 || op > 5);
    cin.ignore();
    return op;
}

int main()
{
    // Crear un objeto lista de teléfonos
    CListaTfnos listatfnos;

    int opcion = 0;

    do
    {
        opcion = menu();

        switch (opcion)
        {
            case 1: // buscar
                buscar(listatfnos, false);
                break;
            case 2: // buscar siguiente
                buscar(listatfnos, true);
                break;
            case 3: // añadir
                anyadir(listatfnos);
                break;
            case 4: // eliminar
                eliminar(listatfnos);
                break;
        }
    }
    while(opcion != 5);
}

```

La siguiente función se ejecuta para las opciones 1 y 2. Permite buscar un elemento que contenga *cadenabuscar*, subcadena que se obtiene del teclado, y si se encuentra, muestra sus datos. Cuando el segundo parámetro de esta función es **true**, permite buscar el siguiente elemento que contenga la subcadena utilizada en la última búsqueda.

```

void buscar(CListaTfnos& listatfnos, bool buscar_siguiente)
{

```

```
static int pos = -1;
static string cadenabuscar;

if (!buscar_siguiente)
{
    cout << "conjunto de caracteres a buscar: ";
    getline(cin, cadenabuscar);
    // Buscar a partir del principio
    pos = listatfnos.buscar(cadenabuscar, 0);
}
else
    // Buscar el siguiente a partir del último encontrado
    pos = listatfnos.buscar(cadenabuscar, pos+1);
if (pos == -1)
    if (listatfnos.longitud() != 0)
        cout << "búsqueda fallida\n";
    else
        cout << "lista vacía\n";
else
{
    cout << listatfnos.registro(pos).obtenerNombre() << endl;
    cout << listatfnos.registro(pos).obtenerDireccion() << endl;
    cout << listatfnos.registro(pos).obtenerTelefono() << endl;
}
}
```

La siguiente función se ejecuta cuando se elige la opción 3. Obtiene los datos nombre, dirección y teléfono del nuevo elemento a añadir desde teclado, crea un objeto *CPersona* y lo añade.

```
void anyadir(CListaTfnos& listatfnos)
{
    string nombre, direccion;
    long telefono;
    cout << "nombre: "; getline(cin, nombre);
    cout << "dirección: "; getline(cin, direccion);
    cout << "teléfono: "; cin >> telefono;
    listatfnos.anyadir(CPersona(nombre, direccion, telefono));
}
```

La siguiente función se ejecuta cuando se elige la opción 4. Obtiene del teclado el número de teléfono correspondiente al objeto *CPersona* que se desea eliminar, busca este objeto y si lo encuentra lo elimina de la lista.

```
void eliminar(CListaTfnos& listatfnos)
{
    long telefono;
    bool eliminado = false;
    cout << "teléfono: "; cin >> telefono;
```

```

eliminado = listatfnos.eliminar(telefono);
if (eliminado)
    cout << "registro eliminado\n";
else
    if (listatfnos.longitud() != 0)
        cout << "teléfono no encontrado\n";
    else
        cout << "lista vacía\n";
}

```

FUNCIONES AMIGAS DE UNA CLASE

El hecho de que un método ordinario sea miembro de una clase implica tres cosas:

1. Tiene acceso al resto de los miembros de la clase, incluso a los miembros privados, lógicamente para manipular el estado de un objeto.
2. Pertenece al ámbito definido por la clase.
3. Debe invocarse para un objeto de su misma clase, al que se refiere por medio del puntero implícito **this**.

En cambio, cuando un método de una clase se declara **static**, sólo participa de los puntos 1 y 2, con la excepción de que sólo puede acceder a los miembros **static**.

Y, finalmente, una función externa no participa de ninguno de los puntos. No obstante, existen casos, como veremos en el capítulo dedicado a operadores sobrecargados, en los que se hace necesario que una función externa participe del punto 1. Esto lo permite C++ declarando la función amiga de la clase.

Para declarar una función externa amiga de una clase, hay que incluir su declaración en el cuerpo de la clase, precedida por la palabra reservada **friend**.

Por ejemplo, en el programa *test.cpp* que escribimos anteriormente en este mismo capítulo para trabajar con objetos *CVector*, vimos que la función *fnVisualizar* no podía acceder a los miembros privados de la clase *CVector*. Según lo expuesto, este problema podría resolverse declarando la función *fnVisualizar* amiga (**friend**) de la clase *CVector*. Esto es:

```

class CVector
{
    friend void fnVisualizar(CVector& vector);
private:
    double *vector; // puntero al primer elemento de la matriz
    int nElementos; // número de elementos de la MATRIZ
protected:

```

```

    double *asignarMem(int);
public:
    CVector(); // crea un CVector con N de elementos por omisión
    CVector(int ne); // crea un CVector con ne elementos
    CVector(double *, int); // crea un CVector desde una matriz
    CVector(const CVector&); // crea un CVector desde otro
    ~CVector(); // destructor
    double& elemento(int i);
    int longitud() const;
};

```

Ahora la definición de la función *fnVisualizar* podría escribirse como se indica a continuación:

```

void fnVisualizar(CVector& v)
{
    int ne = v.longitud();
    for (int i = 0; i < ne; i++)
        cout << setw(7) << v.vector[i];
    cout << "\n\n";
}

```

Observe ahora que la función *fnVisualizar*, en lugar de utilizar el método *elemento* para acceder a los datos de la matriz, puede acceder directamente al atributo *vector* del objeto *v* por ser amiga de *CVector* (expresión *v.vector[i]*).

Una función **friend** puede declararse en cualquier parte de la clase, aunque generalmente se declaran al principio de la definición de la clase. Recuerde que las palabras clave **public**, **protected** y **private** definen el nivel de protección de los miembros de la clase, y nuestra función **friend** no es un miembro de la clase, razón por la cual no se ve afectada por estas palabras clave.

Una función amiga, puesto que tiene que aparecer en la declaración de la clase, es parte de la interfaz de la misma, tanto como lo es cualquier otro miembro. Por lo tanto, no se transgreden los mecanismos de protección. Es la clase quien concede la amistad, al igual que los demás accesos.

El mecanismo de la amistad es importante por dos causas:

1. Una función puede ser amiga de más de una clase, lo que puede conducir a interfaces más claras.
2. Una función amiga admite que se apliquen a su primer argumento conversiones implícitas o definidas por el usuario, donde los métodos no lo admiten. En el capítulo siguiente veremos que esto es especialmente importante en la sobrecarga de operadores binarios, cuando necesitamos contemplar el caso de

que, aunque no sea el primer operando un objeto de la clase, pero sí se pueda convertir en uno, la operación se ejecute satisfactoriamente.

También es posible declarar un método de una clase *C2* amigo de otra clase *C1*. En este caso, la definición de la clase *C2*, clase que aporta el método que va a ser amigo de la clase *C1*, debe preceder a la definición de la clase *C1*. Por ejemplo:

```
// friend.cpp - Métodos de una clase amigos de otra
#include <iostream>
using namespace std;

/////////////////////////////////////////////////////////////////
class C1; // declaración adelantada de C1

class C2
{
private:
    int nc2;
public:
    void AsignarDato(int n) { nc2 = n; }
    int ObtenerDato(const C1&);
};

class C1
{
friend int C2::ObtenerDato(const C1&);
private:
    int nc1;
public:
    void AsignarDato(int n) { nc1 = n; }
};

int C2::ObtenerDato(const C1& obj)
{
    return obj.nc1 + nc2;
}
/////////////////////////////////////////////////////////////////

int main()
{
    C1 objeto1; // objeto de la clase C1
    C2 objeto2; // objeto de la clase C2
    int dato;

    cout << "Nº entero: "; cin >> dato;
    objeto1.AsignarDato(dato);
    cout << "Nº entero: "; cin >> dato;
    objeto2.AsignarDato(dato);
}
```

```
    cout << "\nResultado: ";
    cout << objeto2.ObtenerDato(objeto1) << endl;
}
```

Observe la declaración adelantada de *C1* antes de la definición de la clase *C2*; indica simplemente que la clase *C1* se define más adelante. Si no realizamos esta declaración anticipada, la declaración `int ObtenerDato(C1&)` daría lugar a un error por hacer referencia a una clase, *C1*, aún no declarada.

Por otra parte, la declaración del método *ObtenerDato* como amigo de *C1* se ha hecho especificando la clase a la que pertenece, ya que de omitir `C2::`, se analizaría como una función externa:

```
friend int C2::ObtenerDato(C1&);
```

Además, si *ObtenerDato* no fuera un método amigo de *C1* no podría acceder al atributo privado *nc1*; tendría que hacerlo a través de un método de la clase.

Algunas veces puede ser también necesario que todos los métodos de una clase *C1* sean amigos de otra clase *C2*. Para permitir esto, hay que declarar a la clase *C1* amiga de la clase *C2* así:

```
class C1
{
    // ...
};

class C2
{
    friend class C1;
    // ...
};
```

PUNTEROS A LOS MIEMBROS DE UNA CLASE

El tipo de una función externa, `void fnx(int *)`, por ejemplo, es `void (int *)`, y un puntero a esa función es de tipo `void (*)(int *)`. Por ejemplo:

```
int x;
void fnx(int *p);
void (*pfn)(int *); // declaración de un puntero a una función
// ...
pfn = fnx;          // pfn apunta a la función fnx
pfn(&x);            // llama a la función fnx
```

En C++, a un puntero a un método no se le puede asignar, sin más, la dirección de un método de una clase. La razón es que un método debe operar sobre un objeto de su clase. Por ello, para poder asignar la dirección de un método a un puntero, el puntero tiene que ser del mismo tipo que el método. El tipo de un método de una clase *C*, *void C::fnx(int *)*, por ejemplo, es *void C::(int *)*, y un puntero a ese método es de tipo *void (C::*)(int *)*.

Según lo expuesto, la notación para un puntero a un miembro de la clase *C* es *C::**; y la notación para acceder a la dirección de un miembro de una clase *C* es *&C::*. Para clarificar lo expuesto, considere una clase *CNotas* para la cual están declarados, entre otros, el método *AsignarNota* y el atributo *nota*, como sigue:

```
class CNotas
{
    private:
        float nota;
    public:
        CNotas(float n = 0): nota(n) {};
        void AsignarNota(float n);
        float ObtenerNota() const;
};
```

Para definir un puntero al método *AsignarNota* haríamos lo siguiente:

```
// Definir el tipo derivado pmet: puntero a un método
typedef void (CNotas::*pmet)(float);

// Definir el puntero pmetAsignarNota:
// puntero al método AsignarNota
pmet pmetAsignarNota = &CNotas::AsignarNota;
```

El identificador *pmetAsignarNota* es un puntero al método *AsignarNota* de la clase *CNotas*.

Igualmente, se podría acceder a la dirección del atributo *nota* como sigue:

```
// ...
// Definir un puntero al atributo nota
float CNotas::*patrNota = &CNotas::nota;
// ...
```

El identificador *patrNota* es un puntero al dato miembro *nota* de la clase *CNotas*.

Para acceder a un miembro de una clase *C* referenciado por un puntero, disponemos de los operadores binarios: *.** y *->**.

El operador `.*` liga su segundo operando, que debe ser un puntero a un miembro, a su primer operando, que debe ser un objeto. La sintaxis es:

*objeto.*puntero_a_miembro*

Por ejemplo, el siguiente código invoca al método *AsignarNota* para el objeto *alumno*:

```
CNotas alumno;
// ...
(alumno.*pmetAsignarNota)(nota);
```

La precedencia de `()` es mayor que la de `.*` y `->*`, por eso son necesarios los paréntesis.

Si el primer operando es un *puntero a un objeto*, entonces utilizaremos el operador `->*`. La sintaxis es:

*puntero_a_objeto->*puntero_a_miembro*

Por ejemplo, el siguiente código invoca al método *AsignarNota* para el objeto referenciado por *palumno*:

```
CNotas *palumno = new CNotas;
// ...
(palumno->*pmetAsignarNota)(nota);
```

No es posible definir un puntero a un miembro declarado **static**, ya que, al no identificarse éste con un objeto particular de su clase, puede ser accedido sin la necesidad de que exista un objeto de la misma.

El siguiente ejemplo ilustra los conceptos expuestos. Su función es crear estática o dinámicamente objetos de la clase *CNotas*, asignarles un valor y finalmente visualizar su contenido.

```
// punteros.cpp - Punteros a miembros de una clase
#include <iostream>
#include <iomanip>
using namespace std;

////////////////////////////////////
// Definición de la clase CNotas
class CNotas
{
private:
    float nota;
```

```

public:
    CNotas(float n = 0): nota(n) {};
    void AsignarNota(float n);
    float ObtenerNota() const;
};

void CNotas::AsignarNota(float n) { nota = n; }

float CNotas::ObtenerNota() const
{
    //return nota;

    // Definir un puntero al dato miembro nota
    float CNotas::*patrNota = &CNotas::nota;

    //return (*this).*patrNota;
    return this->*patrNota;
}
/////////////////////////////////////////////////////////////////

// Prototipos de funciones
void Visualizar(CNotas *);

// Definir el tipo derivado pmet: puntero a un método
typedef void (CNotas::*pmet)(float);

int main()
{
    CNotas alumno;
    CNotas *palumno = new CNotas;
    float nota;

    // Definir los tipos derivados pmetAsignarNota:
    // puntero al método AsignarNota
    pmet pmetAsignarNota = &CNotas::AsignarNota;

    // Introducir datos
    cout << "Nota del alumno: ";
    cin >> nota;
    //alumno.AsignarNota( nota );
    (alumno.*pmetAsignarNota)(nota);

    cout << "Nota del alumno: ";
    cin >> nota;
    //palumno->AsignarNota( nota );
    (palumno->*pmetAsignarNota)(nota);
    // Visualizar el contenido de los objetos
    cout << endl;
    cout << "alumno 1, nota: ";
    Visualizar(&alumno);
    cout << "alumno 2, nota: ";

```

```

    Visualizar(palumno);

    delete palumno;
}

void Visualizar(CNotas *palumno)
{
    cout << fixed << setprecision(2);
    cout << setw(12) << palumno->ObtenerNota() << endl;
}

```

Observe, en el método *ObtenerNota* del programa anterior, que también es posible utilizar la expresión ***this** para referirse al objeto para el cual ha sido llamado dicho método. Lo que no se puede es escribir *return *patrNota* porque el operador ***** no es válido para operandos de tipo *float CNotas::**.

EJERCICIOS RESUELTOS

1. Cuando un alumno accede a la Universidad, se matricula de un conjunto de asignaturas en unos determinados estudios o carrera. Si analizamos este supuesto con la intención de escribir un programa orientado a objetos que permita realizar el seguimiento de las asignaturas de las que un alumno se matricula a lo largo de su estancia en la Universidad para conocer en cada momento su estado actual, podemos llegar a la conclusión de que nuestro programa tiene que manipular alumnos, asignaturas, convocatorias y fechas que darán lugar a las clases *CAlumno*, *CAsignatura*, *CConvocatoria* y *CFecha*. Un alumno se matricula de una o más asignaturas en una fecha determinada y cada vez que se examina de una de ellas consume una convocatoria, teniendo que abandonar los estudios si agota seis convocatorias de una asignatura, lo que nos exige guardar por cada asignatura los datos de cada convocatoria consumida. Nota: se trata de hacer un ejercicio sólo para practicar los conceptos expuestos y no de hacer una aplicación profesional.

La última versión de la clase *CFecha* se expuso al hablar de miembros **static** de una clase y será la que utilizaremos en este programa.

La clase *CConvocatoria* incluye los atributos convocatoria consumida (1 a 6), mes y año en que se consumió y nota obtenida en la misma (0 a 10), así como los métodos necesarios para manipular esta información. Según esto, la declaración de esta clase podría ser más o menos así:

```

// convocatoria.h - Declaración de la clase CConvocatoria
#ifndef _CONVOCATORIA_H_
#define _CONVOCATORIA_H_
#include <string>
#include "fecha.h"

```

```

class CConvocatoria
{
private:
    int convocatoria; // número (1, 2, ...) de convocatoria
    std::string fecha; // seis dígitos (mes y año): mmaaaa
    float nota; // nota obtenida en la convocatoria especificada
public:
    CConvocatoria(int c = 0, float n = 0);
    void asignarConvocatoria(int conv);
    int obtenerConvocatoria();
    void asignarFecha(int mes = 0, int anyo = 0);
    std::string obtenerFecha();
    void asignarNota(float n);
    float obtenerNota();
};
#endif // _CONVOCATORIA_H_

```

El constructor construirá un objeto *CConvocatoria* con los valores convocatoria y nota pasados como argumentos (cero por omisión) y la fecha actual obtenida del sistema. Estos valores serán asignados a sus atributos por medio de los métodos *asignarConvocatoria*, *asignarFecha* y *asignarNota*, con el fin de verificar que se encuentran dentro de los rangos permitidos. Además, el mes y el año de la convocatoria serán reducidos a un valor entero de seis dígitos y almacenado como un **string** en el atributo *fecha*. El resto de los métodos que puede ver en la declaración de la clase permiten obtener los valores de los atributos.

Obsérvese que el constructor utiliza una línea de iniciadores. ¿Y por qué si después se invoca a *asignarConvocatoria*? Porque este método empieza comprobando qué valor tiene el atributo *convocatoria*, y de no iniciarlo tendría un valor impredecible (basura).

Según lo expuesto, la definición de la clase puede ser así:

```

// convocatoria.cpp - Definición de la clase CConvocatoria
#include <iostream>
#include "convocatoria.h"
using namespace std;

CConvocatoria::CConvocatoria(int c, float n) :
convocatoria(c) // iniciar atributos
{
    // Comprobar que los valores c y n son válidos y
    // asignar la fecha actual.
    asignarConvocatoria(c);
    asignarFecha();
    asignarNota(n);
}

```

```
void CConvocatoria::asignarConvocatoria(int conv)
{
    if (convocatoria > 6)
    {
        cerr << "error: convocatorias agotadas\n";
        return;
    }
    if (conv < 0 || conv > 6)
    {
        cerr << "error: convocatoria no válida\n";
        convocatoria = 0;
        return;
    }
    convocatoria = conv;
}

int CConvocatoria::obtenerConvocatoria() const
{
    return convocatoria;
}

void CConvocatoria::asignarFecha(int mes, int anyo)
{
    int d, m, a;
    if (CFecha(1, mes, anyo).fechaCorrecta())
    {
        m = mes; a = anyo;
    }
    else
        CFecha::obtenerFechaActual(d, m, a);
    char sfecha[7];
    sprintf(sfecha, "%.2d%.4d", m, a);
    fecha = string(sfecha);
}

string CConvocatoria::obtenerFecha() const
{
    return fecha;
}

void CConvocatoria::asignarNota(float n)
{
    if (n < 0 || n > 10)
    {
        cerr << "error: nota no válida\n";
        nota = 0;
        return;
    }
    nota = n;
}
```



```
float CConvocatoria::obtenerNota()
{
    return nota;
}
```

La clase *CAsignatura* incluye los atributos, identificador y nombre de la asignatura, fecha en la que el alumno se matriculó de ella por primera vez y una lista (una matriz) de convocatorias inicialmente vacía; cada elemento de esta lista se corresponderá con un objeto *CConvocatoria*. Así mismo, incluye los métodos necesarios para manipular esta información. Según esto, la declaración de esta clase podría ser más o menos así:

```
// asignatura.h - Declaración de la clase CAsignatura
#ifndef _ASIGNATURA_H_
#define _ASIGNATURA_H_
#include <string>
#include <vector>
#include "fecha.h"
#include "convocatoria.h"

class CAsignatura
{
private:
    int ID;           // identificador de la asignatura
    std::string nombre; // nombre de la asignatura
    CFecha fecha;    // primera vez que se realizó la matrícula
    std::vector<CConvocatoria> convocatorias;
public:
    CAsignatura(int id = 999999, std::string nom = "");
    void asignarID(int id);
    int obtenerID() const;
    void asignarNombre(std::string nom);
    std::string obtenerNombre() const;
    bool asignarFecha(CFecha& f);
    const CFecha& obtenerFecha();
    CConvocatoria& obtenerConvocatoria(unsigned int i);
    void anyadirConvocatoria(CConvocatoria& c);
    size_t convocatoriasConsumidas();
};

#endif // _ASIGNATURA_H_
```

El constructor *CAsignatura* construirá un objeto con los valores, identificador y nombre pasados como argumentos (o con los valores por omisión) y la fecha actual obtenida del sistema. Estos valores serán asignados a sus atributos por medio de la lista de iniciadores y de los métodos *asignarID* y *asignarFecha*. Como *asignarID* comprueba si el identificador es 0 o negativo, no es imprescindible iniciar este atributo previamente, operación que se hace en la lista de iniciadores.

El método *asignarFecha* asigna al atributo *fecha* el objeto *CFecha* pasado como argumento si la fecha es correcta. Este método devuelve un valor **true** si la operación se realiza satisfactoriamente y **false** en caso contrario.

El método *obtenerConvocatoria* devuelve una referencia al objeto *i* de la lista de convocatorias de un objeto *CAsignatura*. Si el índice está fuera de los límites de la matriz, devuelve una referencia a un objeto **static** iniciado con los valores por omisión (identificador 999999 y nombre nulo); se utiliza un objeto **static** porque un objeto **auto** sería eliminado al finalizar el método y, lógicamente, no se puede mantener una referencia a un objeto que no existe. El hecho de que se devuelva una referencia es para permitir modificar el objeto *CConvocatoria* referenciado; si se devolviera una copia no podríamos actuar sobre el objeto de la matriz.

Los métodos comentados, y el resto de ellos, se exponen a continuación:

```
// asignatura.cpp - Definición de la clase CAsignatura
#include <iostream>
#include "asignatura.h"

using namespace std;

CAsignatura::CAsignatura(int id, string nom):ID(id), nombre(nom)
{
    asignarID(id);           // para controlar errores
    fecha.asignarFecha();    // fecha actual por omisión
    convocatorias.reserve(6); // reservar espacio para 6 elementos
}

void CAsignatura::asignarID(int id)
{
    if (id < 1)
    {
        cerr << "error: Id no válido\n";
        id = 999999;
    }
    ID = id;
}

int CAsignatura::obtenerID() const
{
    return ID;
}

void CAsignatura::asignarNombre(string nom)
{
    nombre = nom;
}
```

```
string CAsignatura::obtenerNombre() const
{
    return nombre;
}

bool CAsignatura::asignarFecha(CFecha& f)
{
    bool b = f.fechaCorrecta();
    if (b) fecha = f;
    return b;
}

const CFecha& CAsignatura::obtenerFecha()
{
    return fecha;
}

CConvocatoria& CAsignatura::obtenerConvocatoria(unsigned int i)
{
    static CConvocatoria c;
    size_t n = convocatoriasConsumidas();

    if ( n == 0 ) return c;
    --i; // ajustar la convocatoria (1, 2, ...)
        // a los subíndices (0, 1, ...) del vector

    if (i >= 0 && i < n)
        return convocatorias[i];
    else
    {
        cerr << "error: convocatorias consumidas " << n << endl;
        return c;
    }
}

void CAsignatura::anyadirConvocatoria(CConvocatoria& c)
{
    convocatorias.push_back(c);
}

size_t CAsignatura::convocatoriasConsumidas()
{
    return convocatorias.size();
}
```

La clase *CAlumno* incluye los atributos DNI, nombre y dirección de un alumno y una lista (una matriz) de asignaturas inicialmente vacía; cada elemento de esta lista se corresponderá con un puntero a un objeto *CAsignatura*. Así mismo,

incluye los métodos necesarios para manipular esta información. Según esto, la declaración de esta clase podría ser más o menos así:

```
// alumno.h - Declaración de la clase CALumno
#ifndef _ALUMNO_H_
#define _ALUMNO_H_
#include <string>
#include <vector>
#include "asignatura.h"

class CALumno
{
private:
    long DNI;
    std::string nombre;
    std::string direccion;
    std::vector<CAsignatura *> asignatura;
public:
    CALumno(long dni = 0, std::string nom = "", std::string dir="");
    CALumno(const CALumno&);
    ~CALumno();
    CALumno& operator=(const CALumno&);
    void asignarDNI(long dni);
    long obtenerDNI() const;
    void asignarNombre(std::string&);
    std::string obtenerNombre() const;
    void asignarDireccion(std::string&);
    std::string obtenerDireccion() const;
    CAsignatura *obtenerAsignatura(unsigned int i);
    bool estaEnActa(int id, int& pos);
    void anyadirAsignatura(CAsignatura *);
    size_t numeroAsignaturas() const;
};

#endif // _ALUMNO_H_
```

El constructor *CALumno* construirá un objeto con los valores DNI, nombre y dirección pasados como argumentos (o con los valores por omisión). Estos valores serán asignados a sus atributos por medio de la lista de iniciadores y, además, invocará al método *asignarDNI* para verificar que DNI no es un valor negativo, por lo que no resulta imprescindible iniciar este atributo previamente en la lista de iniciadores.

Esta clase constituye un ejemplo claro de la necesidad que hay de redefinir el constructor copia y el operador de asignación por omisión. ¿Por qué? Porque tiene un atributo que es una matriz de punteros, y tanto el constructor copia como el operador de asignación copiarán este atributo en el objeto destino, pero no duplicará los objetos apuntados, con la problemática que esto supone según estudiamos

anteriormente en este mismo capítulo. El constructor copia lo implementaremos para que haga la copia del objeto origen, el pasado como argumento, en el objeto destino (**this*) invocando al operador de asignación. El operador de asignación primero eliminará los objetos *CAsignatura* del objeto *CAlumno* destino, después eliminará todos los elementos de la propia matriz dejándola con cero elementos y, finalmente, copiará todos los atributos del origen en el destino, pero duplicando los objetos *CAsignatura* referenciados por el origen.

Esta clase también requiere un destructor que libere la memoria de los objetos *CAsignatura* referenciados por la matriz *asignatura*.

El método *obtenerAsignatura* devuelve un puntero al objeto *i* de la lista de asignaturas de un objeto *CAlumno*. Si el índice está fuera de los límites de la matriz, devuelve un 0. El hecho de que se devuelva un puntero es para permitir modificar el objeto *CAsignatura* referenciado; si se devolviera una copia no podríamos actuar sobre el objeto de la matriz.

El método *anyadirAsignatura* recibe como argumento un puntero al objeto *CAsignatura* a añadir a la lista de asignaturas de un objeto *CAlumno*. Evidentemente, tendremos que añadir una copia ya que si el objeto pasado es **auto** será eliminado cuando el flujo de ejecución salga fuera del ámbito donde fue creado y si fue dinámico, será destruido en el mismo módulo donde se creó (los objetos deben ser destruidos por quien los crea, que es quien dispone de toda la información necesaria, sin necesidad de tener que realizar suposiciones).

El método *estaEnActa* devuelve **true** si el alumno está en el acta de la asignatura *id* cuyo identificador se pasa como argumento; en otro caso devuelve **false**. Un alumno se considera incluido en el acta de una asignatura identificada por *id* si se ha matriculado, no ha aprobado y no ha consumido el total de convocatorias. Cuando se cumplen estos requisitos, el segundo argumento (pasado por referencia) devuelve la posición (0, 1, ...) de la asignatura en la lista de asignaturas del objeto *CAlumno* para el que fue invocado el método.

Los métodos comentados, y los no comentados por ser triviales, se exponen a continuación:

```
// alumno.cpp - Definición de los métodos de la clase CAlumno
#include <iostream>
#include "alumno.h"
using namespace std;

CAlumno::CAlumno(long dni, string nom, string dir):
DNI(dni), nombre(nom), direccion(dir)
{
    asignarDNI(dni);
```

```
    asignatura.reserve(10); // espacio para 10 objetos CAsignatura
}

CALumno::CALumno(const CALumno& x)
{
    *this = x;
}

CALumno::~~CALumno()
{
    for (unsigned int i = 0; i < numeroAsignaturas(); i++)
        delete asignatura[i];
}

CALumno& CALumno::operator=(const CALumno& x)
{
    // Eliminar las asignaturas del objeto CALumno destino (*this)
    for (unsigned int i = 0; i < asignatura.size(); i++)
        delete asignatura[i];
    // Redimensionar la matriz asignatura del destino (*this)
    asignatura.resize(x.asignatura.size());

    // Copiar el alumno origen, x, en el alumno destino
    DNI = x.DNI;
    nombre = x.nombre;
    direccion = x.direccion;
    for (unsigned int i = 0; i < x.asignatura.size(); i++)
        asignatura[i] = new CAsignatura(*(x.asignatura[i]));

    return *this;
}

void CALumno::asignarDNI(long dni)
{
    if (dni < 0)
    {
        cerr << "error: DNI no válido\n";
        dni = 0;
    }
    DNI = dni;
}

long CALumno::obtenerDNI() const
{
    return DNI;
}

void CALumno::asignarNombre(string& nom)
{
    nombre = nom;
}
```

```
string CALumno::obtenerNombre() const
{
    return nombre;
}

void CALumno::asignarDireccion(string& dir)
{
    direccion = dir;
}

string CALumno::obtenerDireccion() const
{
    return direccion;
}

CAsignatura *CALumno::obtenerAsignatura(unsigned int i)
{
    if (numeroAsignaturas() == 0) return 0;
    if (i >= 0 && i < numeroAsignaturas())
        return asignatura[i];
    else
    {
        cerr << "error: índice fuera de límites\n";
        return 0;
    }
}

void CALumno::anyadirAsignatura(CAsignatura *asig)
{
    asignatura.push_back(new CAsignatura(*asig));
}

bool CALumno::estaEnActa(int id, int& i)
{
    // En i se devuelve la posición de la asignatura id en la lista
    if (numeroAsignaturas() == 0) return false;

    // Un alumno pertenece al acta de la asignatura id si se ha
    // matriculado, aún no ha aprobado y no excede el número
    // de convocatorias.
    for (i = 0; i < numeroAsignaturas(); i++)
    {
        CAsignatura *asig = obtenerAsignatura(i);
        if (asig->obtenerID() != id) continue;
        int nconv = asig->convocatoriasConsumidas();
        if (nconv > 0)
            if (asig->obtenerConvocatoria(nconv).obtenerNota() >= 5)
                return false;
        if (nconv == 6)
        {
            cerr << "error: convocatorias agotadas\n";
        }
    }
}
```

```
        break;
    }
    return true; // está en el acta.
}
return false; // no está en el acta.
}

size_t CAumno::numeroAsignaturas() const
{
    return asignatura.size();
}
```

Una vez escritas las clases, vamos a escribir un programa que presente un menú con las opciones: matricular alumnos, poner notas, mostrar el expediente de un alumno y salir. La ejecución de este programa será de la forma siguiente:

1. *Matricular*
2. *Poner notas*
3. *Mostrar expediente*
4. *Salir*

Opción: 1

DNI: 111111
Nombre: Alfonso Sánchez
Dirección: La Calzada, Barcelona
Asignaturas:
ID: 1234
Nombre: Fundamentos de program.

¿Otra asignatura? s/n: s

ID: 1235

...

¿Otra asignatura? s/n: n

1. *Matricular*
2. *Poner notas*
3. *Mostrar expediente*
4. *Salir*

Opción: 2

ID asignatura: 1234
Alfonso Sánchez, nota: 4
Beatriz Galindo, nota: 6

...

1. *Matricular*
2. *Poner notas*
3. *Mostrar expediente*
4. *Salir*


```

    Opción: 3
DNI: 111111
Alumno Alfonso Sánchez:
Asignatura           Convocatoria           Nota
Fundamentos de program.      1              4
                               2              7
Estructura de computad.      1              7
Sistemas operativos                NP
...

```

La función **main** de este programa, en respuesta a las opciones del menú mostrado por la función *menu* que se indica a continuación, almacenará los alumnos matriculados en una matriz *alumno* de objetos *CAumno*, permitirá poner las notas por asignatura y mostrará el expediente del alumno solicitado.

```

int menu()
{
    cout << '\n';
    cout << "1. Matricular\n";
    cout << "2. Poner notas\n";
    cout << "3. Mostrar expediente\n";
    cout << "4. Salir\n";
    cout << endl;
    cout << "    Opción: ";
    int op;
    do
        cin >> op;
    while (op < 1 || op > 4);
    cin.ignore();

    return op;
}

int main()
{
    // Crear una matriz de alumnos inicialmente vacía
    vector<CAumno> alumno;
    int opcion = 0, id, dni;

    do
    {
        opcion = menu();

        switch (opcion)
        {
            case 1:
                matricular(alumno);
                break;

```

```

        case 2:
            cout << "ID asignatura: "; cin >> id;
            poner_notas(alumno, id);
            break;
        case 3:
            cout << "DNI: "; cin >> dni;
            mostrar_expediente(alumno, dni);
            break;
    }
}
while(opcion != 4);
}

```

La opción 1 del menú invoca a una función denominada *matricular*, pasando como argumento la matriz donde se almacenan los alumnos que se matriculan. Esta función permite matricular a un alumno cada vez que se invoca. Para ello:

1. Solicita los datos personales de un alumno.
2. Crea ese objeto *CAlumno*.
3. Solicita los datos de una asignatura y crea ese objeto *CAsignatura*.
4. La añade a la lista de asignaturas del alumno creado en 1 (lo que se añade es una copia del objeto *CAsignatura*). Este punto se repetirá para cada una de las asignaturas de las que se tiene que matricular el alumno.
5. Finalmente, añade el alumno a la matriz de alumnos.

Dejamos como trabajo opcional para el lector verificar si el alumno o las asignaturas que se añaden ya existen y cómo proceder en esos casos.

```

void matricular(vector<CAlumno>& v)
{
    // Datos personales del alumno
    int dni;
    string nombre, direc;

    cout << "DNI:          "; cin >> dni; cin.ignore();
    cout << "Nombre:         "; getline(cin, nombre);
    cout << "Dirección:      "; getline(cin, direc);
    CAlumno al(dni, nombre, direc);

    // Asignaturas de las que se va a matricular
    cout << endl;
    cout << "Asignaturas:\n";
    char respuesta;
    do
    {
        CAsignatura *asig = leerDatosAsig();
        al.anyadirAsignatura(asig);
    }
}

```

```

delete asig;
cout << endl;
do
{
    cout << "¿Otra asignatura? s/n: ";
    cin >> respuesta;
    cin.ignore();
}
while(respuesta != 's' && respuesta != 'n');
}
while(respuesta == 's');
// Añadir el alumno que se acaba de matricular
v.push_back(al);
}

CAsignatura *leerDatosAsig()
{
    CAsignatura *asig;
    int id;
    string nombre_as;
    char sID[8];
    do
    {
        cout << "ID:          "; cin >> id; cin.ignore();
        // Simular que el nombre de la asignatura procede de una base
        // de datos
        sprintf(sID, "%.2d", id);
        nombre_as = string("asignatura") + string(sID);
        if (nombre_as.empty()) cout << "ID no válido\n";
    }
    while (nombre_as.empty());
    cout << "Nombre:          " << nombre_as;
    // Añadir una asignatura
    asig = new CAsignatura(id, nombre_as);
    // la fecha por omisión es la actual
    return asig;
}

```

La opción 2 del menú invoca a una función denominada *poner_notas*, pasando como argumento la matriz de alumnos matriculados y el identificador de la asignatura. Esta función permite poner las notas por asignatura. Para ello:

1. Verifica si el alumno está en el acta de esa asignatura.
2. Si está, crea un objeto *CConvocatoria*.
3. Asigna los datos de esa convocatoria y la añade a la lista de convocatorias de la asignatura (lo que se añade es una copia del objeto *CConvocatoria*). Este punto se repetirá para cada uno de los alumnos que estén en el acta de esta asignatura.

```

void poner_notas(vector<CAumno>& v, int id)
{
    if (v.size() == 0)
    {
        cerr << "No hay alumnos matriculados\n";
        return;
    }
    // Poner la nota de la asignatura id, a todos los alumnos
    // matriculados en ella
    for (unsigned int al = 0; al < v.size(); al++)
    {
        int pos; // posición de la asignatura id en la lista
        if (v[al].estaEnActa(id, pos)) // está en actas
        {
            CConvocatoria conv; // convocatoria a añadir
            cout << v[al].obtenerNombre() << ", nota: ";
            float nota; // nota obtenida en esta convocatoria
            cin >> nota; cin.ignore();
            conv.asignarConvocatoria(
                v[al].obtenerAsignatura(pos)->convocatoriasConsumidas()+1);
            conv.asignarFecha(); // por omisión, fecha actual
            conv.asignarNota(nota);
            v[al].obtenerAsignatura(pos)->anyadirConvocatoria(conv);
        }
    }
}

```

La opción 3 del menú invoca a una función denominada *mostrar_expediente*, pasando como argumento la matriz de alumnos matriculados y el DNI del alumno. Esta función permite mostrar el expediente de ese alumno. Para ello:

1. Verifica si el alumno está matriculado.
2. Si está, muestra su expediente; esto es, por cada asignatura en la que se matriculó desde que ingresó en la Universidad, se muestra el nombre de la asignatura, las convocatorias agotadas y la nota en cada convocatoria.

```

void mostrar_expediente(vector<CAumno>& v, int dni)
{
    if (v.size() == 0)
    {
        cerr << "No hay alumnos matriculados\n";
        return;
    }
    CConvocatoria c;
    unsigned int al, i = 0;
    for (al = 0; al < v.size(); al++)
        if (v[al].obtenerDNI() == dni) break;
    if (al == v.size())
    {

```

```

    cerr << "error: no existe un alumno con ese DNI\n";
    return;
}
// Mostrar el expediente del alumno "dni"
cout << "Alumno " << v[a].obtenerNombre() << ": " << endl;
cout << left << setw(25) << "Asignatura"
    << right << setw(15) << "Convocatoria"
    << setw(15) << "Nota" << endl;
for (unsigned int as = 0; as < v[a].numeroAsignaturas(); as++)
{
    CAsignatura *asig = v[a].obtenerAsignatura(as);
    cout << left << setw(25) << asig->obtenerNombre();
    for (i = 1; i <= asig->convocatoriasConsumidas(); i++)
    {
        c = asig->obtenerConvocatoria(i);
        cout << right << setw(15) << c.obtenerConvocatoria()
            << setw(15) << c.obtenerNota() << endl
            << left << setw(25) << "";
    }
    if (asig->convocatoriasConsumidas() == 0)
        cout << right << setw(30) << "NP" << endl;
    cout << endl;
}
}
}

```

EJERCICIOS PROPUESTOS

1. Modificar el programa realizado en el apartado *Ejercicios resueltos* relativo a alumnos, asignaturas y convocatorias, para que incluya una nueva clase denominada *CEstudios*:

```

class CEstudios
{
private:
    int ID;
    string nombre; // nombre de la carrera
    vector<CALumno> alumnos;
    map<int, string> asignatura; // lista de asignaturas
public:
    // ...
};

```

Añadir a esta clase la funcionalidad necesaria para que el comportamiento del programa sea el mismo, pero ahora partiendo de la definición que se indica a continuación en la función **main**. Realizar en el resto del código las modificaciones requeridas por la incorporación de esta nueva clase.

```
CEstudios estudio; // Crear un objeto estudio (carrera a cursar)
```

El constructor de la clase *CEstudios* será el encargado de llenar el mapa *asignatura* con las parejas identificador-nombre de las asignaturas correspondientes a los estudios elegidos (simular que se leen de un medio externo). De esta forma, cuando un alumno se matricule, bastará con solicitar el identificador de la asignatura, el nombre lo obtendremos de esta lista.

2. Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de una variable. Por ejemplo:

$$2x^3 - x + 8.25 \quad \text{más} \quad 5x^5 - 2x^3 + 7x^2 - 3 \quad \text{igual a} \quad 5x^5 + 7x^2 - x + 5.25$$

Cada término del polinomio será representado por una clase *CTermino* y cada polinomio por una clase *CPolinomio*.

La clase *CTermino* tendrá dos atributos privados: *coeficiente* y *exponente*, y los métodos necesarios para permitir al menos:

- Construir un término, iniciado a 0 por omisión.
- Acceder al coeficiente de un término.
- Acceder al exponente de un término.
- Obtener la cadena de caracteres equivalente a un término con el formato siguiente: $\{+|- \} 7x^4$.

La clase *CPolinomio* tendrá un atributo privado (*polinomio*) que será una matriz que almacenará los términos del polinomio, así como los métodos necesarios para permitir al menos:

- Construir un polinomio, inicialmente con cero términos.
- Obtener el número de términos que tiene actualmente el polinomio.
- Asignar un término a un polinomio colocándolo en orden ascendente del exponente. Si el término existe, se sumarán los coeficientes. Si el coeficiente es nulo, no se realizará ninguna operación. Cada vez que se inserte un nuevo término, se incrementará automáticamente el tamaño del polinomio en 1. El método encargado de esta operación tendrá un parámetro de la clase *CTermino*.
- Sumar dos polinomios. El polinomio resultante quedará también ordenado en orden ascendente del exponente.
- Obtener la cadena de caracteres correspondiente a la representación de un polinomio con el formato siguiente: $+ 5x^5 - 1x^1 + 5.25$.

OPERADORES SOBRECARGADOS

El término *operador sobrecargado* se refiere a un operador que es capaz de desarrollar su función en varios contextos diferentes sin necesidad de otras operaciones adicionales. Por ejemplo, la suma $a + b$ en la práctica para nosotros supondrá operaciones comunes diferentes dependiendo de que estemos trabajando en el campo de los números reales o en el campo de los números complejos. Si dotamos al operador $+$ para que, además de sumar reales, permita también sumar complejos, dependiendo esto del tipo de los operandos, entonces diremos que el *operador $+$* está *sobrecargado*.

Como ejemplo de operadores sobrecargados muy conocidos, aunque no nos hayamos parado a pensar en ello, tenemos el operador de inserción ($<<$) y el operador de extracción ($>>$). Estos operadores los hemos venido utilizando junto con los flujos **cout** y **cin** para mostrar datos de tipos primitivos y derivados en la salida estándar, así como para leerlos de la entrada estándar. Otro ejemplo es el operador de asignación ($=$) que cada clase de objetos proporciona por omisión.

SOBRECARGAR UN OPERADOR

C++ provee la facilidad de asociar una función a un operador estándar, con el fin de que la función sea llamada cuando el compilador detecte este operador en un contexto específico. Se dice entonces que el *operador* está *sobrecargado*, porque los conjuntos de objetos sobre los que puede operar son más.

La sintaxis para declarar un operador sobrecargado es la siguiente:

tipo operador operador([*parámetros*]);

donde *tipo* indica el tipo del valor retornado por la función y *operador* es uno de los de la tabla siguiente:

+	-	*	/	%	^	&	
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	
+=	-=	*=	/=	%=	^=	&=	=
<<=	>>=	[]	()	->	->*	new	delete

Los operadores **::** (resolución de ámbito), **.** (selección de un miembro), **.*** (selección de un miembro referenciado por un puntero), **?:** (operador condicional) y los operadores **sizeof** (tamaño de) y **typeid** (identificación del tipo) no se pueden sobrecargar, simplemente por la operación que desarrollan. Tampoco es posible definir nuevos símbolos como operadores; por ejemplo, ****** para la potenciación.

La palabra clave **operator** más un *operador* de la tabla anterior forman el nombre de la función. ¿Y la lista de parámetros? Lo estudiamos a continuación.

Sabemos que un operador unario se aplica sobre un solo operando y que un operador binario se aplica sobre dos operandos. Pues bien, según esto, cuando se sobrecargue un operador unario utilizando una función externa, ésta debe tomar un parámetro, y dos cuando se sobrecargue un operador binario. Por ejemplo:

```
class C
{
    // ...
};

// Declaraciones de funciones externas
C operator-(C);      // menos unario sobre objetos de la clase C
C operator-(C, C);  // menos binario

int main()
{
    C a, b, c; // a, b y c son objetos de la clase C
    b = -c;    // menos unario
    c = a - b; // menos binario
}

C operator-(C x)
{
    // ...
}

C operator-(C x, C y)
{
    // ...
}
```


La sentencia $b = -c$ implícitamente invoca a la función *operator-* con un parámetro y $c = a - b$, a la función *operator-* con dos parámetros. Por lo tanto, la función **main** podría también escribirse así:

```
int main()
{
    C a, b, c;
    b = operator-(c);    // menos unario
    c = operator-(a, b); // menos binario
}
```

Cuando la función que sobrecarga un operador se corresponde con un método de una clase el razonamiento es análogo, pero teniendo en cuenta que ahora existe un parámetro implícito: se trata del objeto para el que es invocado el método. Por lo tanto, en este caso, el método utilizado para sobrecargar un operador unario tendrá cero parámetros explícitos y el utilizado para sobrecargar un operador binario tendrá uno. Por ejemplo:

```
class C
{
    // Métodos
public:
    C operator-();    // operador menos unario
    C operator-(C); // operador menos binario
};
// ...

int main()
{
    C a, b, c; // a, b y c son objetos de la clase C
    b = -c;    // menos unario
    c = a - b; // menos binario
}
```

Obsérvense los métodos de la clase *C*. Ambos toman un argumento implícito: el objeto para el cual son invocados (referenciado por **this**). Entonces, el primer método, por tratarse de un operador unario, toma un solo argumento (el implícito) y el segundo, por tratarse de un operador binario, toma dos argumentos: uno implícito (el que está a la izquierda del operador) y otro explícito (el de la derecha). Según esto, la función **main** anterior podría también escribirse así:

```
int main()
{
    C a, b, c;          // a, b y c son objetos de la clase C
    b = c.operator-(); // menos unario
    c = a.operator-(b); // menos binario
}
```

Cuando se sobrecarga un operador, éste conserva su propiedad de binario o unario y mantiene invariable su prioridad de evaluación y su asociatividad. Por ello se sugiere que se hagan sobrecargas que no realicen una operación diferente a la esperada por el operador utilizado. Esto es, la sobrecarga de un operador debe ser clara y sin ambigüedades; de lo contrario, nuestra forma natural de pensar respecto a la prioridad de operadores nos puede traicionar. Por ejemplo, si sobrecargamos el `*` para realizar la suma de complejos y el `+` para realizar la multiplicación, tendremos que recordar a la hora de utilizarlos que el operador `+` debe tener mayor prioridad que el operador `*`, lo que va en contra de nuestra forma natural de pensar; otro ejemplo, el operador `^` puede ser el más apropiado para la potenciación, pero el hecho de que su prioridad sea más baja que la del resto de los operadores aritméticos desaconseja su utilización en este contexto.

Los operadores `=`, `[]`, `->` y `()` cuando se sobrecarguen deben ser definidos como métodos de una clase y no como funciones externas, lo que asegura que su primer operando sea un *valor-i*; el resto de los operadores no requieren esta exigencia. Un *valor-i* representa una región de almacenamiento que puede aparecer a la izquierda del signo igual (`=`). Generalmente, un *valor-i* es una expresión que referencia a un objeto.

Los operadores sobrecargados son normalmente utilizados con clases, para facilitar determinadas operaciones con los objetos de las mismas. Resultan especialmente útiles cuando se trata de trabajar con tipos abstractos de datos que definen objetos pertenecientes al campo de las Matemáticas; por ejemplo, operaciones con números complejos.

Como ejemplo, vamos a escribir un programa que permita realizar la suma y la resta de números complejos. Un número complejo estará definido por un objeto de la clase *CComplejo* y para realizar las operaciones solicitadas esta clase incluirá un método para sobrecargar el operador `+` y otro para el `-`.

```
// complejo.h - Declaración de la clase CComplejo
#ifndef _COMPLEJO_H_
#define _COMPLEJO_H_

// Clase para operar con números complejos
class CComplejo
{
private:
    double real, imag; // parte real e imaginaria

public:
    CComplejo(double r = 0, double i = 0) : // constructor
        real(r), imag(i)
    {
    }
}
```

```
    double ObtenerParteReal() const { return real; }
    double ObtenerParteImag() const { return imag; }
    void AsignarComplejo(double r, double i);
    CComplejo operator+(CComplejo x); // sumar complejos
    CComplejo operator-(CComplejo x); // restar complejos
};
```

```
#endif // _COMPLEJO_H_
```

```
// complejo.cpp - Definición de la clase CComplejo
```

```
#include "complejo.h"
```

```
// Asignación de complejos
```

```
void CComplejo::AsignarComplejo(double r, double i)
{
    real = r;
    imag = i;
}
```

```
// Suma de complejos
```

```
CComplejo CComplejo::operator+(CComplejo x)
{
    return CComplejo(real + x.real, imag + x.imag);
}
```

```
// Diferencia de complejos
```

```
CComplejo CComplejo::operator-(CComplejo x)
{
    return CComplejo(real - x.real, imag - x.imag);
}
```

```
// test.cpp - Operaciones con números complejos.
```

```
#include <iostream>
```

```
#include "complejo.h"
```

```
using namespace std;
```

```
void visualizar(const CComplejo&);
```

```
int main()
```

```
{
    CComplejo a, b, c(1.5, 2), d;
    double re, im;
```

```
    cout << "Número complejo - escriba re im: ";
```

```
    cin >> re >> im;
```

```
    a.AsignarComplejo(re, im);
```

```
    b = a;
```

```
    d = a + b - c;
```

```
    d = d + CComplejo(3, 3);
```

```
    visualizar(d);
```

```
}
```

```
// Visualizar un complejo
void visualizar(const CComplejo& c)
{
    cout << "(" << c.ObtenerParteReal() << ", ";
    cout << c.ObtenerParteImag() << ")" << endl;
}
```

A continuación realizamos un análisis de cómo trabaja este programa:

```
CComplejo a, b, c(1.5, 2), d;
```

invoca al constructor *CComplejo* para cada uno de los objetos *a*, *b*, *c* y *d* declarados. Los complejos *a*, *b* y *d* son iniciados a 0, por omisión, y el complejo *c* es iniciado con los valores 1.5 y 2.

```
a.AsignarComplejo(re, im);
```

envía el mensaje *AsignarComplejo* al objeto *a*. La respuesta es que se ejecuta el método *AsignarComplejo* que asigna al complejo *a* los valores *re* e *im* pasados como argumentos.

```
b = a;
```

asigna al complejo *b* el valor del complejo *a*. Recuerde que cuando en una clase no se define el operador de asignación, el compilador C++ define uno por omisión. Para la clase *CComplejo*, este operador es así:

```
CComplejo& CComplejo::operator=(const CComplejo& c)
{
    real = c.real; imag = c.imag;
    return *this;
}
```

El hecho de que el método anterior devuelva una referencia al objeto asignado permite realizar asignaciones múltiples; por ejemplo, $a = b = c$.

Cuando se ejecuta la expresión $b = a$, lo que sucede es que implícitamente se invoca al método **operator=** así: $b.operator=(a)$, lo que pone de manifiesto, una vez más, que en una expresión donde interviene un operador binario, el objeto que recibe el mensaje es el que está a la izquierda del operador.

Así mismo, en la clase *CComplejo* observamos que hay dos operadores sobrecargados más: $+$ y $-$. Los prototipos de los métodos correspondientes son:

```
CComplejo operator+(CComplejo x);
```

que es invocado por el operador + cuando sus operandos son complejos, y

```
CComplejo operator-(CComplejo x);
```

que es invocado por el operador – cuando sus operandos son complejos.

Recordar que los métodos de una clase pueden ser invocados solamente para un objeto de esa clase. Según esto, los métodos anteriores tienen dos argumentos: uno implícito, el objeto para el cual es invocado el método, y otro explícito, el especificado en la lista de argumentos del método. Por lo tanto, la expresión:

```
a + b que es equivalente a la llamada a.operator+(b)
```

indica que el complejo *a* recibe el mensaje de sumarse con el complejo *b*. La respuesta a este mensaje es que se ejecuta el método **operator+**, el cual realiza la operación requerida.

Debido a que el primer operando está implícito, un intento de declarar un método con un prototipo como:

```
CComplejo operator+(CComplejo x, CComplejo y);
```

daría lugar a un error, ya que C++ espera un solo argumento.

La declaración anterior puede realizarse, cuando se trate de una función externa. En este caso, para poder utilizarla con objetos de la clase *CComplejo*, habría que declararla amiga de la clase; de lo contrario, no tendría acceso a los datos privados.

Para el operador – seguiríamos un razonamiento análogo. Siguiendo con el análisis, la sentencia:

```
d = d + CComplejo(3, 3);
```

incrementa el complejo *d* en el valor del complejo (3, 3). Para realizar esta operación, la expresión *CComplejo(3, 3)* llama al constructor *CComplejo* que construye un objeto temporal, el complejo (3, 3).

Finalmente, la función externa *visualizar* permite presentar en pantalla el complejo pasado como argumento:

```
visualizar(d);
```

Esta función utiliza los métodos *ObtenerParteReal* y *ObtenerParteImag* para acceder a los atributos de un objeto complejo.

Ahora que ya tenemos claro el concepto de sobrecarga de un operador, conviene resaltar algunas restricciones. No se puede sobrecargar un operador binario (operador que se aplica a dos operandos) para crear un operador unario (operador que se aplica a un solo operando). Igualmente, no se puede sobrecargar un operador unario para realizar operaciones binarias. Por supuesto, los operadores que pueden actuar como unarios y binarios se pueden sobrecargar para utilizarlos en uno u otro contexto. Otra restricción importante es que aunque sea posible modificar la definición de un operador (por ejemplo, sobrecargar el operador \wedge para que realice la potenciación), no es posible modificar su precedencia (prioridad).

UNA CLASE PARA NÚMEROS RACIONALES

A modo de ejemplo, vamos a construir una clase *CRacional* que almacena un número como cociente de dos enteros. O sea, un número racional es un número representado por el cociente de dos números enteros (lo que normalmente llamamos quebrado), como $5/7$. El número de la izquierda se denomina numerador y el de la derecha, denominador.

Esta clase es útil porque muchos números no pueden ser representados exactamente utilizando el tipo **float**. Por ejemplo, $1/3 + 1/3 + 1/3$, que es 1 , utilizando el tipo **float** sería $0,333333 + 0,333333 + 0,333333$, que es $0,999999$. La clase *CRacional* que escribimos a continuación evita este tipo de errores.

En el ejercicio planteado identificamos una entidad que es el número racional. Entonces, *numerador* y *denominador* son atributos de la clase que hemos decidido llamar *CRacional*:

```
class CRacional
{
    private:
        long numerador;
        long denominador;
    public:
        // Métodos
};
```

Pensemos ahora en el conjunto de operaciones que podemos realizar con los números racionales (a modo de ejemplo, sólo veremos algunas de las varias posibles):

1. Construir un número racional. El constructor implícito por omisión (sin argumentos) no es suficiente, ya que para formar un racional requerimos dos argumentos: numerador y denominador. Por ello definiremos explícitamente un constructor.

2. Operaciones de asignación.
3. Operaciones aritméticas. Suma, resta, multiplicación y división.
4. Comparación de dos números racionales. Igual, menor y mayor.
5. Operaciones de entrada y salida.
6. Incremento, decremento y cambio de signo.

Empecemos con el constructor. La construcción de un número racional sin argumentos parece lógico que sea la construcción del racional $0/1$. Partiendo de este supuesto, el prototipo del constructor *CRacional* puede ser:

```
CRacional(long = 0, long = 1); // constructor
```

Otras operaciones que debe realizar el constructor es verificar si el denominador es 0, en cuyo caso forzamos a que sea 1, o negativo, en cuyo caso invertimos el signo del numerador y del denominador, y simplificar la fracción si es posible. Según esto la definición del constructor *CRacional* puede ser así:

```
CRacional::CRacional(long num, long den):
    numerador(num), denominador(den)
{
    if (den == 0) denominador = 1;
    if (den < 0)
    {
        numerador = -numerador;
        denominador = -denominador;
    }
    Simplificar();
}
```

El método *Simplificar* lo declaramos protegido. Por lo tanto, una primera aproximación a la declaración de la clase podría ser así:

```
class CRacional
{
private:
    long numerador;
    long denominador;
protected:
    CRacional& Simplificar();
public:
    CRacional(long = 0, long = 1); // constructor
    // Métodos para operaciones de asignación
    // Métodos para operaciones aritméticas
    // Métodos para operaciones de relación
    // Métodos para operaciones de E/S
```

```

    // Métodos para operaciones sobre un único operando
};

```

Además del constructor declarado explícitamente, la clase *CRacional* define por omisión un constructor copia. Siempre que sea posible, es preferible utilizar este constructor ya que son los compiladores, y no nosotros, los que conocen perfectamente el comportamiento por omisión. Este constructor simplemente copia todos los atributos del objeto pasado como argumento y en general es así:

```

CRacional::CRacional(const CRacional& c) :
numerador(c.numerador), denominador(c.denominador)
{
}

```

SOBRECARGA DE OPERADORES BINARIOS

A continuación vamos a estudiar cómo sobrecargar los operadores de asignación y aritméticos, los de relación y los de E/S. Dentro de este conjunto de operadores los hay que modifican la estructura interna del objeto, como +=, y operadores que producen un nuevo objeto, como +.

Sobrecarga de operadores de asignación

En este conjunto de operadores distinguimos, además del operador =, otros como +=, /=, etc.; por ejemplo, partiendo de las siguientes declaraciones:

```
CRacional a, b(4, 16);
```

para asignar a un racional *a* otro racional *b*, sólo tenemos que escribir:

```
a = b;
```

La operación anterior no requiere escribir ningún método, ya que si una clase no define el operador de asignación, el compilador C++ define uno por omisión. Por ejemplo, para la clase *CRacional*, este método sería así:

```

CRacional& CRacional::operator=(const CRacional& c)
{
    numerador = c.numerador;
    denominador = c.denominador;
    return *this;
}

```

El hecho de que el método anterior devuelva una referencia al objeto asignado permite realizar asignaciones múltiples; por ejemplo, $a = b = c$, que explícitamente

te se escribiría así: $a.operator=(b.operator=(c))$. Se puede observar que esta expresión es válida porque $b.operator=(c)$ devuelve un objeto *CRacional* (se trata del operando de la derecha de $=$). No olvide que la asociatividad de los operadores de asignación es de derecha a izquierda.

Analicemos ahora los operadores de asignación compuesta; por ejemplo, el operador $+=$:

```
CRacional a, b(4, 16), c;
a += b; // equivale a: a = a + b;
```

Para realizar la operación del ejemplo anterior es necesario definir el operador $+=$ para los números racionales. La solución es sobrecargar este operador. Se trata de un método, **operator+=**, con un parámetro declarado como una referencia a un objeto constante *CRacional* (el operando de la derecha), que devuelve un valor *CRacional*. El otro objeto implicado en la suma (el operando de la izquierda) es aquél para el cual se invoca el método; este operando, a su vez, almacenará el resultado. De acuerdo con lo expuesto, la definición de este método puede ser así:

```
CRacional& CRacional::operator+=(const CRacional& r)
{
    numerador = numerador * r.denominador +
                denominador * r.numerador;
    denominador = denominador * r.denominador;
    return (*this).Simplificar();
}
```

El hecho de que el parámetro del método sea una referencia es simplemente por una cuestión de eficacia; esto es, de esta forma se evita una llamada al constructor copia para copiar el objeto pasado como argumento, y una llamada al destructor para eliminar este objeto local (la copia) cuando el método finalice; por otra parte, declarar el objeto constante impide que el argumento pasado se modifique, característica implícita cuando el argumento se pasa por valor.

El valor ***this** devuelto por el método retorna el objeto para el cual fue invocado, pero con los nuevos valores de *numerador* y *denominador*. Para simplificar el objeto ***this**, se invoca al método *Simplificar*.

El método *Simplificar* utiliza el algoritmo de Euclides para obtener el máximo común divisor (*mcd*) del numerador y del denominador y para simplificar el número racional dividiendo el numerador y el denominador por este *mcd*.

```
CRacional& CRacional::Simplificar()
{
    // Máximo común divisor. Algoritmo de Euclides
    long mcd, temp, resto;
```

```
mcd = labs( numerador );
temp = denominador;

while ( temp > 0 )
{
    resto = mcd % temp;
    mcd = temp;
    temp = resto;
}

// Simplificar
if ( mcd > 1 )
{
    numerador /= mcd;
    denominador /= mcd;
}
return *this;
}
```

Devolver una referencia al objeto ***this** evita también una llamada al constructor copia para crear un objeto temporal necesario para una siguiente operación (casi siempre una asignación) y una llamada al destructor para eliminar este objeto temporal cuando el método finalice. En este mismo supuesto, devolver un objeto en lugar de una referencia, si se trata de un *objeto local*, el que se llame o no al constructor copia depende del compilador utilizado y de las optimizaciones que éste realice; puede ser que el compilador decida no destruir el objeto local para no tener que crear otro objeto temporal, copia de éste, hasta que realice la operación donde fue requerido.

Las siguientes líneas de código invocan al método **operator+=**, para realizar la suma de dos números racionales:

```
CRacional a(1, 2), b(1, 3), c;
a += b; // equivale a: a = a.operator+(b)
```

Cuando se ejecute la sentencia $a += b$, el valor de a , que era $1/2$, será ahora $5/6$ y el valor de b no habrá cambiado.

Sobrecarga de operadores aritméticos

Los operadores aritméticos, como $+$, producen un nuevo objeto a partir de sus dos operandos. Para definirlos podemos proceder de dos formas: escribiendo los métodos para que realicen las operaciones a partir de los atributos o bien para que las realicen invocando a otros métodos ya implementados.

Como ejemplo, a continuación se expone la sobrecarga del operador `+` desde estos dos puntos de vista.

```
const CRacional CRacional::operator+(const CRacional &r)
{
    CRacional temp(numerador * r.denominador +
                  denominador * r.numerador,
                  denominador * r.denominador);
    return temp;
}
```

Esta versión crea un objeto local *temp* invocando al constructor *CRacional* con los valores resultantes de la suma y devuelve *temp* como resultado una vez simplificado. Se puede observar que en este caso no es necesario invocar al método *Simplificar* puesto que ya lo hace el constructor. El hecho de devolver un objeto **const** evita que el compilador acepte una sentencia como $a + b = c$. Otra forma de escribir este método es:

```
const CRacional CRacional::operator+(const CRacional &r)
{
    return CRacional(numerador * r.denominador +
                    denominador * r.numerador,
                    denominador * r.denominador);
}
```

Esta versión crea un objeto temporal invocando al constructor *CRacional* con los valores resultantes de la suma y devuelve como resultado dicho objeto, una vez simplificado por el propio constructor.

Esta otra versión que presentamos a continuación invoca al constructor copia para crear un objeto *temp* copia del primer operando (el implícito), después invoca al operador `+=` y suma *temp* con el segundo operando, el resultado se almacena en *temp*, y finalmente devuelve el resultado.

```
const CRacional CRacional::operator+(const CRacional& r)
{
    CRacional temp = *this;
    return temp += r;
}
```

Éste es un caso típico donde si en lugar de devolver el objeto devolviéramos una referencia al objeto, podrían darse resultados inesperados. Obsérvese que *temp* es un objeto local al método; quiere esto decir que será eliminado por el destructor de la clase cuando finalice el mismo, lo que daría lugar a que la referencia fuera a un objeto que ya no existe.

Aritmética mixta

Recordar que en el capítulo anterior vimos que un constructor que puede ser invocado con un único parámetro (puede ser un constructor con un único parámetro o un constructor con todos sus parámetros con valores por omisión) implícitamente especifica una conversión del tipo de su parámetro al tipo del constructor. Por ejemplo:

```
CRacional r = 3; // equivale a: CRacional r = CRacional(3);
```

Este ejemplo construye el racional 3/1 a partir de un entero 3. El atributo *denominador* toma el valor especificado por omisión en el constructor *CRacional*.

Según lo expuesto, supongamos ahora que uno de los operandos que intervienen en la suma es un entero. Por ejemplo:

```
CRacional a(1, 2), c;
int b = 3;
c = a + b; // equivale a: c = a.operator+(b)
```

Cuando ejecutemos este código, observaremos que todo funciona correctamente. Esto se debe a que el compilador, utilizando el constructor de la clase, intenta convertir el argumento pasado al método, que es el entero *b*, en un objeto *CRacional* y si es posible, el resultado será satisfactorio. Como el constructor de la clase tiene argumentos por omisión, todo ha funcionado correctamente. Pero ahora supongamos que se ejecuta este otro código:

```
CRacional a(1, 2), c;
int b = 3;
c = b + a; // equivale a: c = b.operator+(a)
```

En este caso el compilador muestra un error. Obsérvese la notación funcional: $c = b.operator+(a)$; está claro que el método *operator+* no puede ser invocado para un objeto que no sea de su clase. Esto significa que el operando de la izquierda tiene que ser un objeto de la clase *CRacional*.

Puesto que la suma es conmutativa y nuestro operador *+* no se comporta de esa forma, la solución, según hemos visto, está en que el compilador, utilizando el constructor de la clase, realice la conversión de los argumentos implicados en la suma cuando sea preciso, lo que sugiere pasar de forma explícita ambos operandos como argumentos. Esto es:

```
c = operator+(a, b);
```

Según lo expuesto al principio de este capítulo, esto supone implementar una función externa como la siguiente (añadir el prototipo de esta función al fichero *racional.h*).

```
const CRacional operator+(const CRacional& r1, const CRacional& r2)
{
    CRacional temp = r1;
    return temp += r2;
}
```

Otra forma de implementar la función anterior es la siguiente:

```
const CRacional operator+(const CRacional& r1, const CRacional& r2)
{
    return CRacional(r1.numerador * r2.denominador +
                    r1.denominador * r2.numerador,
                    r1.denominador * r2.denominador);
}
```

Ahora bien, para que una función externa pueda acceder a los atributos de una clase, hay que declararla amiga de la clase (**friend**).

```
class CRacional
{
    friend const CRacional operator+(const CRacional&, const CRacional&);
    // ...
}
```

Una función amiga, puesto que tiene que aparecer en la declaración de la clase, es parte de la interfaz de la misma, tanto como lo es cualquier otro miembro. Por lo tanto, no se transgreden los mecanismos de protección. Es la clase quien concede la amistad, al igual que los demás accesos.

Si el constructor de la clase no puede convertir el argumento pasado a un objeto de su clase porque no existe una conversión implícita, en este caso de ese argumento a **long**, entonces será necesario redefinir el operador + utilizando una función externa adecuada para lograr que este operador acepte operandos de los tipos deseados. Por ejemplo, para permitir

```
CRacional a(1, 2), c;
double x = 2.0;
c = x + a;
```

se necesita redefinir el operador + para que acepte un primer operando de tipo **double**:

```
CRacional operator+(const double d, const CRacional& r)
{
```

```

// Conversión explícita de double a long truncando la
// parte fraccionaria si existe.
CRacional temp = static_cast<long>(d);
return temp += r;
}

```

Sobrecarga de operadores de relación

Otras operaciones de uso frecuente con números racionales son las operaciones de relación. Por ejemplo, si quisiéramos saber si dos números racionales *a* y *b* son iguales, lo más sencillo sería escribir una sentencia como la siguiente:

```

CRacional a, b(4, 16);
if (a == b) ...

```

Pero sucede que el operador `==` no está definido para los números racionales. La solución es sobrecargar este operador. En principio, se trata de un método **operator==** con un parámetro que es una referencia a un objeto constante *CRacional*, que devuelve un valor **true** o **false**. El otro objeto implicado en la comparación es aquél para el cual se invoca el método. Pero, siguiendo un razonamiento similar al efectuado para la función **operator+**, llegamos a la misma conclusión; este método tiene que escribirse como una función externa amiga de la clase.

```

bool operator==(const CRacional& r1, const CRacional& r2)
{
    return  r1.numerador * r2.denominador ==
           r1.denominador * r2.numerador;
}

```

El resto de las operaciones de relación se desarrollan de forma similar a la expuesta.

Métodos adicionales

El método anterior podría escribirse de otra forma si dotamos a la clase de otros dos métodos que accedan al *numerador* y al *denominador*, respectivamente, según se muestra a continuación:

```

class CRacional
{
    // ...
public:
    long Numerador() const { return numerador; }
    long Denominador() const { return denominador; }
    // ...
};

```

Como estos métodos no modifican el valor de un racional se pueden declarar **const**. Utilizando estos métodos podemos escribir esta otra versión de **operator==** que no necesita declararse amiga de la clase:

```
bool operator==(const CRacional& r1, const CRacional& r2)
{
    return r1.Numerador() * r2.Denominador() ==
           r1.Denominador() * r2.Numerador();
}
```

También, utilizando los métodos *Numerador* y *Denominador*, podríamos escribir una función externa *Visualizar* que muestre un número racional:

```
void Visualizar(const CRacional& r)
{
    cout << r.Numerador() << "/";
    cout << r.Denominador() << endl;
}
```

Aunque los métodos *Numerador* y *Denominador* no permiten que otras funciones tengan acceso directo a la representación de un racional, tienen un pequeño inconveniente, y es que por ser métodos públicos cualquier usuario tiene acceso a ellos; por lo tanto, podría escribir código en base a los valores del *numerador* y del *denominador*. Puesto que el objetivo es implementar una clase para operar con números racionales, no parece muy aceptable lo anterior, sobre todo cuando existen otras alternativas que no necesitan de los métodos *Numerador* y *Denominador* según demuestra la primera versión que escribimos de **operator==**. Análogamente, podemos estudiar cómo sobrecargar el operador de inserción (<<) para mostrar un racional en la salida estándar. Ídem con respecto al operador de extracción (>>) para aceptar un racional desde la entrada estándar.

Sobrecarga del operador de inserción

C++ tiene una clase denominada **ostream**, obtenida a partir de la plantilla **basic_ostream**, que incluye varias sobrecargas del operador << para la salida de valores de tipos predefinidos.

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_ostream : virtual public basic_ios<Ch, Tr>
{
    // ...
public:
    // ...
    basic_ostream<...>& operator<<(short);
    basic_ostream<...>& operator<<(double);
    // ...
}
```

```

    basic_ostream<...>& put(Ch c);
};

typedef basic_ostream<char>                ostream;

```

Así mismo, otras funciones **operator<<** que toman como argumento uno o más caracteres se implementan como funciones externas utilizando el método **put**. Por ejemplo:

```

template <class Ch, class Tr>
basic_ostream<Ch, Tr>& operator<<<(
    basic_ostream<Ch, Tr>&, const char *
);

```

En todos los casos, el método **operator<<**, tras ejecutar un código que imprimirá el dato especificado, retorna una referencia al objeto de tipo **ostream** para el que fue invocado (normalmente al objeto predefinido **cout**), de forma que el operador de inserción pueda encadenarse; esto permite insertar en dicho objeto **ostream** no un solo dato, sino varios. Por ejemplo, una expresión similar a

```
cout << d1 << " " << d2;
```

será interpretada como

```
operator<<<(cout.operator<<<(d1), " ").operator<<<(d2);
```

Después de este estudio, la pregunta que surge es: ¿podríamos utilizar la misma sintaxis para visualizar un objeto de una clase definida por el usuario? Por ejemplo, para mostrar un objeto *r* de la clase *CRacional*, ¿podríamos hacerlo mediante el siguiente código?

```
CRacional r(1, 2);
cout << r;
```

La expresión `cout << r` del ejemplo anterior podría ser interpretada por el compilador C++ de alguna de las dos formas siguientes:

1. `cout.operator<<<(r)`, si existiera un método **operator<<** con un parámetro de tipo *CRacional* en la clase del objeto **cout**.
2. `operator<<<(cout, r)`, si existiera una función externa **operator<<** con dos parámetros: uno del tipo de **cout** y otro de tipo *CRacional*.

Como ninguna de esas sobrecargas existen, tendremos nosotros que implementar una, pero, ¿dónde y de qué forma? En la clase **ostream** de la biblioteca de

C++ resultaría un tanto complicado. En el cuerpo de la clase *CRacional* no es posible porque un método de una clase tiene que ser invocado para un objeto de la misma y **cout** no es un objeto de la clase *CRacional*. Por lo tanto, tendremos que escribir una función externa, y si además necesitamos que esta función acceda a los atributos de la clase *CRacional*, tendremos que declararla amiga de esta clase:

```
class CRacional
{
    friend ostream& operator<<(ostream& os, const CRacional& c);
    // ...
};
```

Obsérvese que *os* es una referencia a un objeto de tipo **ostream**, en este caso a **cout**, y que *c* es una referencia al objeto *CRacional* que deseamos mostrar. A su vez, esta función retornará una referencia al objeto **ostream** para la que fue invocada, de forma que el operador de inserción pueda encadenarse como se muestra a continuación:

```
cout << r1 << r2 << ...
```

En la expresión anterior se supone que *r1*, *r2*, ... son objetos de tipo *CRacional*. Según lo expuesto, para mostrar un número racional añadiremos al fichero *racional.cpp* la función siguiente:

```
// Mostrar un número racional
ostream& operator<<(ostream& os, const CRacional& r)
{
    return os << r.numerador << "/" << r.denominador;
}
```

En el siguiente ejemplo se observa que la forma de utilizar el operador de inserción que acabamos de implementar no difiere de lo que ya conocemos:

```
int main()
{
    CRacional a(1, 2), c;
    int b = 3;

    c = a + b;
    cout << a << " + " << b << " = " << c << endl;
}
```

La sentencia `cout << a << " + " << b << " = " << c << endl` invoca a la función **operator<<** una vez por cada objeto especificado. Para saber qué sobrecarga de la función **operator<<** se invoca, el compilador compara los argumentos en la llamada (operandos a la izquierda y a la derecha del operador de inserción)

con los parámetros formales de la función a fin de emparejarlos. De acuerdo con esto,

```
cout << a // equivale a: operator<<(cout, a)
```

llama a la función externa `operator<<(ostream& os, const CRacional& r)` implementada en `racional.cpp`, la cual devuelve una referencia a **cout** que se aplica a la siguiente operación de inserción, que es

```
cout << " + "
```

que llama a la función externa `operator<<(ostream&, const char *)` de la biblioteca de C++, la cual devuelve una referencia a **cout** que se aplica a la siguiente operación de inserción, que es

```
cout << b
```

que llama al método `operator<<(int)` de **ostream**, el cual devuelve una referencia a **cout** que se aplica a la siguiente operación de inserción; y así sucesivamente.

Este mismo proceso se aplica a la expresión que forma el cuerpo de la función externa **operator<<** amiga de *CRacional*:

```
os << r.numerador << "/" << r.denominador
```

Sobrecarga del operador de extracción

C++ tiene una clase denominada **istream**, obtenida a partir de la plantilla **basic_istream**, que incluye varias sobrecargas del operador `>>` para la entrada de valores de tipos predefinidos.

```
template <class Ch, class Tr = char_traits<Ch> >
class basic_istream : virtual public basic_ios<Ch, Tr>
{
    // ...
public:
    // ...
    basic_istream<...>& operator>>(int&);
    basic_istream<...>& operator>>(double&);
    // ...
};
```

Igual que en el apartado anterior apoyándonos en la clase **ostream** escribimos un operador de inserción para la clase *CRacional*, podemos también ahora definir un operador de extracción para la misma clase, apoyándonos en la clase **istream**

definida en el fichero *<istream>*. Quiere esto decir que los conceptos expuestos para el operador `<<` en el apartado anterior también son aplicables al operador `>>`.

Según esto, la implementación de una función externa **operator>>** para leer un número racional de la forma *entero*, o *entero/entero*, puede ser la siguiente:

```
class CRacional
{
    friend istream& operator>>(istream&, CRacional&);
    // ...
};

// Asignar un número racional
istream& operator>>(istream& is, CRacional& r)
{
    long num = 0, den = 1;
    char car = '\0';

    cout << "(entero[/entero]) ";
    is >> num;          // leer el numerador
    while(is.fail()) // mientras el dato sea incorrecto
    {
        is.clear();
        is.ignore(numeric_limits<int>::max(), '\n');
        cout << "(entero[/entero]) ";
        is >> num;
    }
    if (is.peek() != '\n') // si hay denominador, leerlo
    {
        is >> car;
        if (car == '/')
            is >> den;
        else
            is.clear(ios::badbit); // activar el indicador de error
    }
    if (is) r = CRacional(num, den); // llamar al constructor
    return is;
}
```

En el siguiente ejemplo se observa que la forma de utilizar el operador de extracción que acabamos de implementar no difiere de lo que ya conocemos:

```
int main()
{
    CRacional a, b, c;

    cin >> a >> b;
    c = a + b;
    cout << a << " + " << b << " = " << c << endl;
```

```
}
```

Siguiendo un razonamiento análogo al que hicimos para el operador de inserción,

```
cin >> a
```

llama a la función externa `operator>>(istream& is, CComplejo& c)` amiga de la clase `CRacional`, la cual devuelve una referencia al objeto `cin` que se aplica a la siguiente operación de extracción; y así sucesivamente.

La función `peek` devuelve el siguiente carácter sin extraerlo del flujo.

SOBRECARGA DE OPERADORES UNARIOS

La sobrecarga de un operador unario es similar a la de un operador binario. Pensemos, por ejemplo, en la operación de *negación* y supongamos la siguiente declaración:

```
CRacional a;
```

Si quisiéramos verificar si el objeto `a` es o no 0, haciendo uso de la sintaxis de C++ seguramente escribiríamos algo así:

```
if (!a) // equivale a: if (a.operator!())  
    cout << "número racional nulo\n";
```

Pero sucede que el operador `!` no está definido para los números racionales. La solución es sobrecargar este operador. Se trata de un método sin parámetros explícitos que devuelve un valor `true` o `false`. El objeto implicado en esta operación es aquél al que se le aplica el operador `!`. Entonces, este método puede escribirse de la forma siguiente:

```
bool CRacional::operator!()  
{  
    return !numerador; // devuelve true o false  
}
```

Incremento y decremento

Los operadores `++` y `--` son los únicos operadores C++ que pueden utilizarse como prefijo o como sufijo sobre un operando. Además, cuando el resultado de una expresión que utiliza estos operadores se asigna a una variable, el valor de ésta será diferente en función de que dichos operadores se hayan utilizado como prefi-

jo o como sufijo, lo que significa que su comportamiento es diferente. Consecuentemente, cuando sobrecarguemos estos operadores, tendremos que definir dos métodos, uno para cada caso.

Por ejemplo, supongamos la siguiente sentencia:

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

Un método que realice la operación anterior ($++a$) tiene que retornar el valor de a incrementado. Entonces, cuando sobrecarguemos este operador en una clase como *CRacional*, el método correspondiente no tendrá parámetros explícitos por tratarse de un operador unario y devolverá el objeto *CRacional* para el que fue invocado incrementado en una unidad. Según lo expuesto, este método puede escribirse de la forma siguiente:

```
// Operador ++ como prefijo
CRacional CRacional::operator++()
{
    numerador += denominador;
    return *this;
}
```

En cambio, si el operador $++$ se utiliza como *sufijo*, el valor del objeto será también incrementado en una unidad, pero el método **operator++** devolverá el valor del objeto sin incrementar. Para distinguir entre las sobrecargas como prefijo y como sufijo, C++ dispuso en este último caso que el método tuviera un parámetro de tipo **int** que nunca será utilizado; simplemente sirve para saber que el operador se está utilizando como sufijo. Según esto, este método puede escribirse así:

```
// Operador ++ como sufijo
CRacional CRacional::operator++(int)
{
    CRacional temp = *this;
    numerador += denominador;
    return temp;
}
```

Observe que el objeto devuelto es el mismo que invoca al método.

Resumiendo, una expresión como $++a$ equivale a la llamada $a.operator++()$ y una expresión como $a++$ equivale a la llamada $a.operator++(int)$.

Operadores unarios/binarios

Un operador como `-` puede utilizarse indistintamente como operador unario o como operador binario. Como ya sabemos, el método **operator-** como operador unario no tiene parámetros explícitos y devuelve un objeto *CRacional* del mismo valor que el que invoca al método, pero de signo contrario. Por ejemplo:

```
CRacional a, b, c;
c = -a + b; // -a equivale a la llamada a.operator-()
```

De acuerdo con lo expuesto, este método puede escribirse de la forma siguiente:

```
// Operador - unario
CRacional CRacional::operator-()
{
    CRacional temp(-numerador, denominador);
    return temp;
}
```

Obsérvese que el objeto para el que se invoca el método no se modifica. Quiere esto decir que un método como el siguiente no sería correcto porque modificaría el objeto al que se le aplica el operador `-` unario:

```
CRacional CRacional::operator-()
{
    numerador = -numerador;
    return *this;
}
```

La función **operator-** como operador binario tiene la misma forma y explicación que la función **operator+** vista anteriormente.

```
// Operador - binario
const CRacional operator-(const CRacional& r1, const CRacional& r2)
{
    return CRacional(r1.numerador * r2.denominador -
                    r1.denominador * r2.numerador,
                    r1.denominador * r2.denominador);
}
```

CONVERSIÓN DE TIPOS DEFINIDOS POR EL USUARIO

Hay dos tipos de conversiones: implícitas, las cuales son realizadas automáticamente por el compilador, y explícitas, las cuales fuerzan una determinada conver-

sión utilizando una construcción *cast*. Las situaciones de conversión que se exponen a continuación son de un tipo básico a otro tipo también básico y las realiza el compilador implícitamente:

- Cuando se asigna un valor. Por ejemplo:

```
long a;
int b = 10;
a = b; // el valor de b se convierte a long
```

- Cuando se ejecuta una operación aritmética. Por ejemplo:

```
float a = 10.5, c;
int b = 5;
c = a + b; // el valor de b se convierte a float
```

- Cuando se pasa un argumento a una función. Por ejemplo:

```
int a = 2;
double b = logaritmo(a);
// ...

float logaritmo(float x) // el valor de a se convierte a float
{
    // ...
}
```

- Cuando se retorna un valor desde una función. Por ejemplo, según se ha definido anteriormente la función *logaritmo*,

```
double b = logaritmo(a); // el valor devuelto por logaritmo se
                        // convierte a double
```

En cambio, cuando trabajamos con clases, por tratarse de tipos definidos por el usuario, tenemos nosotros mismos que construir las conversiones que deseamos que realice el compilador cuando utilice un objeto de alguna de ellas. Estas conversiones pueden ser entre clases o entre una clase y un tipo predefinido. Para ello disponemos de dos mecanismos: *constructores* y *operadores de conversión*. Tales conversiones, denominadas normalmente *conversiones definidas por el usuario*, se realizan implícitamente; igual que las conversiones estándar.

Por ejemplo, una función que espera un argumento de tipo *C* puede ser invocada no sólo con un argumento de tipo *C*, sino también con un argumento de tipo *X* si existe una conversión de *X* a *C*.

Conversión mediante constructores

Una conversión puede definirse mediante un *constructor* que acepte un argumento de un determinado tipo y lo convierta en un objeto de su clase. Antes, al escribir la clase *CRacional*, hemos visto un ejemplo. Veamos otro partiendo de la definición de la clase *CComplejo* expuesta al principio de este capítulo; por ejemplo, podemos incluir en la misma un constructor como el siguiente:

```
class CComplejo
{
private:
    double real, imag; // parte real e imaginaria
public:
    CComplejo(double r = 0, double i = 0) : // constructor
        real(r), imag(i)
    {
    }
    CComplejo(int r) : real(r), imag(0) {}
    // ...
};
```

Este constructor no solamente permite iniciar un objeto *CComplejo* a partir de un valor entero, sino que también permite asignar directamente un **int** a un objeto *CComplejo* (vea en el capítulo anterior el calificador **explicit**). Por ejemplo:

```
CComplejo c(3); // Construye el complejo (3, 0).
c = 6;         // Equivale a c = CComplejo(6),
               // construye el complejo (6, 0).
```

Este ejemplo utiliza el constructor *CComplejo* de un solo argumento para convertir implícitamente un entero en un objeto *CComplejo*. La sentencia $c = 6$ llama al constructor y convierte el entero en un objeto *CComplejo* temporal, y a continuación lo asigna al objeto c .

Este tipo de conversión puede realizarse igualmente utilizando el constructor con argumentos por omisión que se indica a continuación; por lo tanto, no sería necesario añadir el constructor anterior:

```
CComplejo(double r = 0, double i = 0) : // constructor
    real(r), imag(i)
{
}
```

En este caso, cuando se ejecuta la sentencia $c = 6$, se llama al constructor realizándose la conversión estándar del entero 6 a **double**, y a continuación se cons-

truye un objeto *CComplejo* temporal iniciándose sus atributos *real* e *imag* con los valores *r* e *i*; después, el objeto resultante se asigna a *c*.

El resultado final es que se pueden realizar operaciones como la siguiente:

```
CComplejo a, b(3, 4);
a = b + 5;
```

En este caso, cuando se ejecuta la sentencia $a = b + 5$, primero se llama al constructor para construir el complejo $(5, 0)$, para lo cual se realiza la conversión estándar del entero 5 a **double** y después se ejecuta el cuerpo del constructor que construye un objeto *CComplejo* temporal. A continuación se llama al método **operator+**, que invocando al constructor construye y devuelve un objeto temporal resultado de $b + (5, 0)$:

```
CComplejo CComplejo::operator+(CComplejo x)
{
    return CComplejo(real + x.real, imag + x.imag);
}
```

Finalmente se asigna al complejo *a* el objeto temporal resultante de la suma.

Operadores de conversión

Según lo expuesto en el apartado anterior, observamos que el uso de un constructor para especificar la conversión entre tipos predefinidos y/o definidos por el usuario es cómodo, pero hay conversiones que no se pueden especificar mediante este mecanismo, como una conversión de un tipo definido por el usuario a un tipo predefinido, ya que un tipo predefinido no es una clase donde podamos añadir un constructor, o bien una conversión de un objeto de una clase nueva a otro de una clase perteneciente a una determinada biblioteca, ya que tendríamos que modificar esta clase de la biblioteca para poder añadir un constructor.

Para solucionar estos problemas C++ proporciona *operadores de conversión*. Por ejemplo, supongamos que necesitamos que se realice de una forma implícita, o explícita si existe ambigüedad, la conversión de un *CRacional* a un **double** según se puede observar en el ejemplo siguiente:

```
double d;
CRacional r(1, 2);
d = r; // r tiene que convertirse a double
```

Para que la sentencia $d = r$ se ejecute correctamente, es preciso que ocurra una conversión implícita de *CRacional* a **double**. Este tipo de conversión no se

puede realizar con un constructor, ya que **double** no es una clase en la que podamos definir un constructor. Pero sí se puede realizar mediante un *operador de conversión*.

La sintaxis para definir un operador de conversión es la siguiente:

C::operator T ();

Un método *operator T()* de una clase *C*, donde *T* es un nombre de tipo, define una conversión de *C* al tipo especificado por *T* (tipo predefinido o tipo definido por el usuario). Como ejemplo, vamos a dotar a la clase *CRacional* de un operador de conversión a **double**.

```
class CRacional
{
    // ...
private:
    long numerador;
    long denominador;
protected:
    CRacional& Simplificar();
public:
    CRacional(long = 0, long = 1); // constructor
    operator double(); // operador de conversión CRacional a double
    // ...
};

// ...

// Conversión de CRacional a double
CRacional::operator double()
{
    return static_cast<double>(numerador)/denominador;
}
```

Obsérvese que el método *operator double* convierte un objeto *CRacional*, representado por los atributos *numerador* y *denominador*, a un valor **double** dado por el cociente *numerador/denominador*. Ahora, cada vez que aparezca un *CRacional* donde se necesite un **double**, se utilizará el método *operator double* para realizar la conversión.

Un *operador de conversión* no puede ser **static**, no puede tener argumentos ni tipo del valor retornado. El tipo del valor retornado está implícito en el nombre del método. La declaración de este operador se hace en el tipo fuente para:

- permitir que objetos de esa clase sean convertidos a un tipo predefinido, o
- permitir que objetos de esa clase sean convertidos a objetos de otra clase.

Un *operador de conversión* puede llamarse explícitamente, pero su principal utilidad es que, igual que un constructor, sea llamado automáticamente por el compilador cuando la evaluación de una expresión requiere el tipo de conversión realizado por él. Por ejemplo:

```
double d;
CRacional r(1, 2);
d = r.operator double(); // llamada explícita al método
d = static_cast<double>(r); // conversión explícita
d = r; // conversión implícita
```

El compilador puede ejecutar simultáneamente conversiones estándar y conversiones definidas por el usuario. Por ejemplo:

```
int i = r;
```

En este ejemplo, primero se convierte el número racional *r* a **double**, utilizando el operador correspondiente de conversión, y a continuación se realiza una conversión estándar de **double** a **int** truncando la parte fraccionaria.

Un ejemplo más; al exponer anteriormente en este mismo capítulo la sobrecarga del operador de extracción, apareció una expresión como la siguiente:

```
if (is) r = CRacional(num, den);
```

donde *is* era una referencia al objeto **cin** de la clase **istream**. En esta sentencia, la expresión *is* es una llamada implícita al operador de conversión **operator void *** definido en la plantilla **basic_ios** así:

```
template<typename _CharT, typename _Traits>
class basic_ios : public ios_base
{
    // ...
    operator void*() const
    {
        return this->fail() ? 0 : const_cast<basic_ios*>(this);
    }
    // ...
}
```

Obsérvese que **operator void *** devuelve un 0 si la última vez que se utilizó el flujo para el que fue invocado, **cin** en nuestro caso, ocurrió un error; en otro caso, devuelve un valor distinto de 0.

También puede definirse un operador de conversión que convierta un objeto de una clase en otro de otra clase. Como ejemplo, a continuación se define un operador de conversión de *CRacional* a *CComplejo*.

```
class CRacional
{
    // ...
    private:
        long numerador;
        long denominador;
    protected:
        CRacional& Simplificar();
    public:
        CRacional(long = 0, long = 1); // constructor
        operator double(); // conversión CRacional a double
        operator CComplejo(); // conversión CRacional a CComplejo
        // ...
};

// ...

// Conversión de CRacional a double
CRacional::operator double()
{
    return (double)numerador/((double)denominador);
}

// Conversión de CRacional a CComplejo
CRacional::operator CComplejo()
{
    return CComplejo((double)numerador/((double)denominador);
}
```

El método *operator CComplejo* convierte un objeto *CRacional* en otro de la clase *CComplejo*. Por ejemplo:

```
CComplejo c, b(3, -2);
CRacional r(1, 2);

c = b + r; // conversión implícita de r a CComplejo
```

En este ejemplo *r* es un número racional, por lo que al realizarse la operación $b + r$, previamente será llamado implícitamente el método *operator CComplejo* para convertir *r* a *Complejo*, realizándose a continuación la suma.

Ambigüedades

En programas con clases que definen muchos caminos de conversión, el compilador prueba y hace implícitamente lo que puede y de la forma más simple posible. En caso de que no pueda decidir, presenta un mensaje de error. Por eso debemos evitar el definir dos o más caminos de conversión entre un tipo y otro, ya que pueden dar lugar a ambigüedades.

Esto es, definir una conversión mediante un constructor de un tipo fuente a un tipo destino y un operador de conversión del tipo destino al tipo fuente dará lugar a ambigüedades; por eso es aconsejable definir uno u otro, pero no ambos. Por ejemplo, la inclusión del operador de conversión *operator double* en la clase *CRacional* en algunos casos crea problemas de ambigüedad, ya que existe un constructor que permite la conversión inversa. Como ejemplo, vamos a analizar el siguiente código:

```
CRacional r(3, 7), c;
double d = 3;
c = r + d;
```

Puesto que el operador `+` además de estar definido para sumar números reales también se ha sobrecargado para sumar números racionales, el compilador no tiene criterios para elegir si convierte el **double** a *CRacional* utilizando el constructor y realiza la suma de dos números racionales, o convierte el *CRacional* a **double** utilizando el método *operator double* y realiza la suma de dos números reales (el resultado sería convertido a *CRacional*), razón por la que genera un error. La forma más sencilla de solucionar este problema es especificar una conversión explícita de uno de los operandos al tipo del otro. Por ejemplo:

```
c = r + static_cast<CRacional>(d); // suma dos números racionales
```

ASIGNACIÓN

Siempre que construyamos una clase, C++, además de crear por omisión un *constructor copia* para esa clase, sobrecarga también el *operador de asignación* (**operator=**) para permitir copiar objetos de esa clase; la copia la realiza miembro a miembro. No obstante, en algunas ocasiones esto no será suficiente y tendremos que redefinir dicho operador. Por ejemplo, volvamos a la clase *CVector* expuesta en el capítulo anterior.

La sobrecarga del operador de asignación que C++ añade a la clase *CVector* es así:

```

CVector& CVector::operator=(const CVector& v)
{
    nElementos = v.nElementos;
    vector = v.vector;
    return *this;
}

```

Supongamos ahora que desde una función cualquiera de un programa se ejecuta el siguiente código:

```

CVector vector1(10), vector2(5);
// ...
vector2 = vector1; // equivale a: vector2.operator=(vector1);

```

El resultado de la asignación `vector2 = vector1` es que `vector` y `v.vector` apuntan a la misma matriz y quizás no sea esto lo que deseamos; además, la memoria referenciada por `vector2` no ha sido liberada. Vea la exposición que se hace para el constructor copia en el apartado *Punteros como atributos de una clase* en el capítulo anterior. Seguramente nuestra intención era que `vector2` fuera un duplicado de `vector1`. Esto exige redefinir el operador de asignación así:

```

// Operador de asignación
CVector& CVector::operator=(const CVector& v)
{
    nElementos = v.nElementos;           // número de elementos
    delete [] vector;                    // borrar la matriz actual
    vector = asignarMem(nElementos);     // crear una nueva matriz
    for (int i = 0; i < nElementos; i++)
        vector[i] = v.vector[i];        // copiar los valores
    return *this;                        // permitir asignaciones encadenadas
}

```

Obsérvese que el operador de asignación toma una referencia a un objeto `CVector`. Para realizar la asignación, primero copia el atributo *número de elementos*. A continuación libera la memoria referenciada por el vector destino y asigna memoria para una nueva matriz con un número de elementos igual al número de elementos de la matriz del vector origen. Por último se hace la copia desde la matriz origen a la matriz destino, elemento a elemento.

Ahora, la línea `vector2 = vector1` copiará el `vector1` en el `vector2` duplicando la matriz apuntada por `vector`. Observe que `vector1` tiene 10 elementos y `vector2` tiene cinco elementos; de ahí que el operador de asignación, antes de realizar la copia de los elementos de la matriz de `vector1`, borre la matriz actual de `vector2` y asigne memoria para una nueva matriz de las dimensiones de `vector1`.

INDEXACIÓN

El operador de indexación, `[]`, se considera un operador binario. Permite acceder a los objetos de las clases como si fuesen matrices unidimensionales. Cuando se sobrecargue este operador, el método `operator[]` debe ser definido como un método de una clase y no como una función externa, lo que asegura que su primer operando sea un objeto. Por tratarse de un operador binario, tiene dos parámetros: uno explícito de cualquier tipo y otro implícito que se corresponde con el objeto para el que se invoca el método.

La sintaxis para utilizar este operador es la misma que empleamos para acceder a los elementos de una matriz unidimensional. Esto es, suponiendo un objeto x de una clase C que defina el método `operator[]`, una expresión con subíndice $x[y]$ es interpretada como $x.operator[](y)$. El argumento y puede ser de cualquier tipo, lo que permite que entidades como las matrices asociativas se puedan definir con la notación convencional.

Por ejemplo, volviendo a la clase `CVector` implementada en el capítulo anterior, recordamos que incluía un método `elemento` para permitir el acceso al elemento i de un objeto `CVector`. Este método estaba definido así:

```
double& CVector::elemento(int i)
{
    return vector[i];
}
```

Esto permitía a las aplicaciones que utilizaban esta clase implementar su propia función para visualizar un objeto `CVector`. Por ejemplo:

```
void fnVisualizar(CVector& vector)
{
    int ne = vector.longitud();
    for (int i = 0; i < ne; i++)
        cout << setw(7) << vector.elemento(i);
    cout << "\n\n";
}
```

Puesto que `vector` representa una matriz, parece más lógico utilizar la notación `vector[i]` que `vector.elemento(i)`. Según lo expuesto anteriormente, esa notación la podremos utilizar sobrecargando el operador de indexación en la clase `CVector`, lo que supone sustituir el método `elemento` por este otro:

```
double& CVector::operator[](int i)
{
    return vector[i];
}
```

Ahora, una función que utilice el método anterior mostrará una forma más natural y familiar de acceso a los elementos de una matriz. Por ejemplo:

```
void fnVisualizar(CVector& v)
{
    int ne = v.longitud();
    for (int i = 0; i < ne; i++)
        cout << setw(7) << v[i];
    cout << "\n\n";
}
```

La expresión $v[i]$ equivale a la llamada $v.operator[](i)$. Fíjese que este método con respecto a su antecesor sólo ha cambiado en el nombre. El hecho de que devuelva una referencia permite utilizarlo a la izquierda y a la derecha del operador de asignación.

LLAMADA A FUNCIÓN

El operador función, **()**, se considera un operador binario. Es similar al operador de indexación visto en el apartado anterior. Cuando se sobrecargue este operador, el método **operator()** debe ser definido como un método de una clase y no como una función externa, lo que asegura que su primer operando sea un objeto. A diferencia del operador de indexación, puede no tener parámetros explícitos o tener varios de cualquier tipo.

La sintaxis para utilizar este operador es la misma que empleamos para llamar a una función. Esto es, suponiendo un objeto x de una clase C que incluya el método **operator()**, una expresión $x(y, z)$ es interpretada como $x.operator()(y, z)$.

Se puede observar que el método **operator()** con un parámetro explícito puede sustituir al método **operator[]**. Por lo tanto, el aplicar uno u otro dependerá del contexto en el que se desarrolle la operación. Ahora bien, cuando el número de parámetros que se necesiten sea distinto de uno, tendremos que utilizar obligatoriamente el método **operator()**.

El uso más importante del operador **()** es proporcionar a los objetos que de alguna manera se comportan como funciones una sintaxis habitual de llamada a función. También su uso es importante como operador de indexación para matrices multidimensionales.

A continuación se muestra un ejemplo que implementa una clase *CIniciar* cuyos objetos se comportan como una función que permite iniciar un objeto *CVector* con un valor predeterminado.

```
// iniciar.h - clase CIniciar
// Iniciar un CVector con un determinado valor
#include "vector.h"
```

```
class CIniciar
{
private:
    double val;
public:
    CIniciar(double x = 0);
    void operator()(CVector& v);
    void valor(double x = 0);
};
```

```
// iniciar.cpp - Definición de la clase CIniciar
#include "iniciar.h"
```

```
CIniciar::CIniciar(double x)
{
    val = x;
}

void CIniciar::operator()(CVector& v)
{
    for (int i = 0; i < v.longitud(); i++)
        v[i] = val;
}

void CIniciar::valor(double x)
{
    val = x;
}
```

Un objeto *CIniciar* se inicia con un **double** y cuando se le invoca utilizando el operador **()**, inicia el objeto *CVector* pasado como argumento con ese valor **double**. El siguiente ejemplo, que utiliza las clases *CVector* y *CIniciar*, crea un vector y lo inicia con el valor 1.

```
int main()
{
    CVector vector1(5); // objeto CVector
    CIniciar iniciar(1); // objeto CIniciar iniciado con el valor 1
    iniciar(vector1); // equivale a: iniciar.operator()(vector1)
    fnVisualizar(vector1);
}
```

Ejecución:

1 1 1 1 1

DESREFERENCIA

El operador desreferencia, **operator->**, se considera un operador unario. Cuando se sobrecargue este operador, el método **operator->** debe ser definido como un método de una clase y no como una función externa, lo que asegura que su primer operando sea un objeto. Por tratarse de un operador unario no tiene argumentos.

La sintaxis para utilizar este operador es la misma que la empleada para acceder a un miembro de una estructura referenciada por un puntero. Esto es, suponiendo un objeto x de una clase C que incluya el método **operator->**, una expresión $x \rightarrow y$ es interpretada como $(x.operator \rightarrow ()) \rightarrow y$. Según la interpretación de $x \rightarrow y$, **operator->()** debe devolver un puntero a un objeto de una clase o a un miembro de su clase siempre y cuando sea de un tipo definido por el usuario.

La sobrecarga de este operador es especialmente útil cuando se trabaja con objetos que actúan como punteros y que además realizan alguna acción sobre el objeto accedido a través de ellos.

Por ejemplo, se puede definir una clase *Ptr_CVector* para el acceso a objetos *CVector*. El constructor *Ptr_CVector* tomará un **int** que se utilizará para crear una matriz de punteros a objetos *CVector*. La dirección de esta matriz y su número de elementos se almacenarán en una estructura de tipo *matriz_d*. Un objeto *Ptr_CVector* actuará como un puntero a esta estructura, para lo cual habrá que sobrecargar el operador **->** de *Ptr_CVector*.

```
// ptr_vector.h - clase Ptr_CVector
// Matriz de punteros a objetos CVector
#include "vector.h"

class Ptr_CVector
{
private:
    struct matriz_d
    {
        CVector **pVector;
        int nElementos;
    } p;
public:
    Ptr_CVector(int n = 1);
    ~Ptr_CVector();
    matriz_d *operator->() { return &p; }
```

```

    CVector*& operator[](int i) { return p.pVector[i]; }
};

// ptr_vector.cpp - Definición de la clase Ptr_CVector
#include <iostream>
#include "ptr_vector.h"
using namespace std;

Ptr_CVector::Ptr_CVector(int n)
{
    p.pVector = new CVector *[n];
    p.nElementos = n;
    fill(p.pVector, p.pVector+n, static_cast<CVector *>(0));
}

Ptr_CVector::~Ptr_CVector()
{
    delete [] p.pVector;
}

```

El siguiente ejemplo, que utiliza las clases *CVector*, *CIniciar* y *Ptr_CVector*, crea un objeto *vector* que encapsula una matriz de punteros a objetos *CVector*. Obsérvese que *vector* actúa como un puntero.

```

int main()
{
    CIniciar iniciar; // objeto CIniciar

    Ptr_CVector vector(5);
    for (int i = 0; i < vector->nElementos; i++)
    {
        vector[i] = new CVector(i+1);
        iniciar.valor(i);
        iniciar(*vector[i]);
        fnVisualizar(*vector[i]);
    }

    for (int i = 0; i < vector->nElementos; i++)
        delete vector[i];
}

```

Ejecución:

```

0
1    1
2    2    2
3    3    3    3
4    4    4    4    4

```

SOBRECARGA DE LOS OPERADORES `new` y `delete`

Una clase puede gestionar por sí misma el espacio de memoria libre que necesite para la creación de objetos sobrecargando los operadores `new` y `delete`.

El operador `new` llama a la función `operator new` y el operador `delete` llama a la función `operator delete`. Cuando se sobrecarguen estos operadores, la definición de estas funciones deberá satisfacer una serie de requerimientos que exponemos a continuación.

Sobrecarga del operador `new`

Cuando en un programa se ejecuta una sentencia como alguna de las siguientes,

```
int *pint = new int;           // objeto de tipo int
int *pmint = new int[TMAX];  // matriz de objetos de tipo int
```

se invoca, en el primer caso, a la función `operator new` y en el segundo, a la función `operator new[]`, que asignan espacios de memoria de tamaños `sizeof(int)` y `sizeof(int)*TMAX` bytes, respectivamente. Además, cuando el tipo de datos para los que se ha reservado ese espacio se corresponde con una clase de objetos, una llamada al operador `new` provoca una llamada al constructor de esa clase por cada uno de los objetos que se vayan a ubicar. Si no hay suficiente espacio de memoria para la asignación requerida, la función lo notificará (vea el operador `new` en el capítulo 9).

Cuando el operador `new` se utiliza para asignar memoria para un objeto de una clase que no sobrecarga este operador, o para una matriz de objetos, se invoca al operador `new` global (`::new`). Este operador está definido de la forma:

```
void *operator new( size_t tamaño );
void *operator new[]( size_t tamaño );
```

En cambio, cuando el operador `new` se utiliza para asignar memoria para un objeto de una clase `C` que define una sobrecarga para este operador, o para una matriz de objetos, se invoca al método `operator new` de dicha clase. En este caso, el prototipo del método tiene que coincidir con alguno de los siguientes:

```
void * C::operator new( size_t tamaño );
void * C::operator new( size_t tamaño[, tipo p1] [, tipo p2] ... );
void * C::operator new[]( size_t tamaño );
void * C::operator new[]( size_t tamaño[, tipo p1] [, tipo p2] ... );
```

En este caso, una declaración como $C *p = new C$; invocará a la función con un argumento y una declaración como $C *p = new(x, y) C$; invocará a una función con tres argumentos (*tamaño* = `sizeof(C)`, $p1 = x$ y $p2 = y$). Cuando se declara el operador **new** en una clase, se oculta la definición del operador **new** global.

Un método **operator new** definido en una clase es un método estático (aun cuando no se declare explícitamente **static**, por lo tanto no puede ser virtual), debe devolver **void *** y tener un primer argumento de tipo **size_t** (tipo definido en el fichero *cstddef*). Este argumento recibe automáticamente un valor en bytes igual al tamaño del bloque de memoria a asignar. La razón por la que tiene que ser estático es porque se invoca antes del constructor, por lo tanto no opera sobre un objeto de la clase. En lugar de ello, asigna el bloque de memoria sobre el que el constructor creará el objeto. Esto quiere decir que hay un grado de separación entre la asignación y la iniciación que redundará en programas más claros.

Cuando se ejecuta **new**, primero se busca una definición para este operador en la clase del objeto a crear; si no se encuentra, entonces se busca en sus clases base, y por último, si no se encuentra, se ejecuta el operador **new** global.

Como ejemplo, retomemos la clase *CVector* implementada en el capítulo anterior y añadamos a la misma las sobrecargas mostradas a continuación:

```
void *CVector::operator new(size_t tam)
{
    return asignarMem(tam, 0);
}

void *CVector::operator new[](size_t tam)
{
    return asignarMem(tam, 0);
}

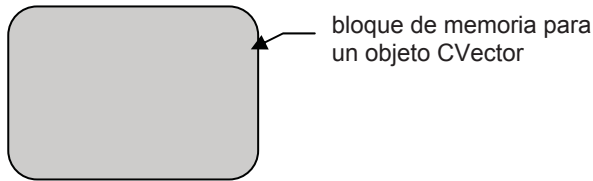
void *CVector::asignarMem(size_t tam, char car)
{
    void *p = malloc(tam);
    if (p == 0)
    {
        cout << "Insuficiente espacio de memoria\n";
        exit(-1);
    }
    // Iniciar el bloque de memoria con el valor car
    fill(static_cast<char *>(p), static_cast<char *>(p)+tam, car);
    return p;
}
```

La primera sobrecarga del operador **new** será invocada cada vez que se cree un objeto *CVector* dinámicamente y la segunda, cuando se cree una matriz diná-

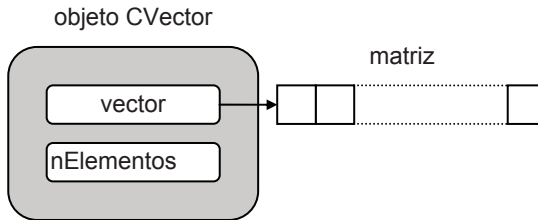
mica de objetos *CVector*. Para probar ambas sobrecargas, vamos a escribir un programa que incluya una función **main** como la escrita a continuación:

```
int main()
{
    const int n = 3;
    CVector *pvector1 = new CVector;
    CVector *pvector2 = new CVector[n];
    // ...
}
```

Este ejemplo construye un objeto *CVector* apuntado por *pvector1* y una matriz de *n* objetos *CVector* apuntada por *pvector2*. Cuando se construye el objeto *pvector1*, primero se invoca al método *operator new(size_t)* para asignar el bloque de memoria sobre el que se construirá el objeto:



y después se invoca al constructor *CVector()* para construir el objeto; esto es, sobre el espacio de memoria anteriormente asignado se toman *sizeof(int)* bytes para almacenar el entero correspondiente al número de elementos de la matriz de tipo **double**, y *sizeof(double *)* bytes para almacenar la dirección del bloque de memoria que el constructor reserva para los *nElementos* de la matriz.



Cuando se construye la matriz de objetos *pvector2*, primero se invoca al método *operator new[](size_t)* para asignar el bloque de memoria sobre el que se construirán los objetos elementos de la matriz y después se invoca al constructor *CVector()* una vez por cada objeto de la matriz.

Sobrecarga del operador delete

Cuando en un programa se ejecuta una sentencia como alguna de las siguientes,

```
delete pvector1;
delete [] pvector2;
```

se invoca, en el primer caso, a la función **operator delete** y en el segundo, a la función **operator delete[]**, que liberan el espacio de memoria asignado por **new**. Además, cuando el tipo de los datos se corresponde con una clase de objetos, una llamada al operador **delete** provoca una llamada al destructor de esa clase por cada uno de los objetos que se elimina.

Cuando el operador **delete** se utiliza para liberar la memoria de un objeto de una clase que no sobrecarga este operador, o de una matriz de objetos, se invoca al operador **delete** global (**::delete**). Este operador está definido de la forma:

```
void operator delete( void * );
```

En cambio, cuando el operador **delete** se utiliza para liberar la memoria para un objeto de una clase *C* que define una sobrecarga para este operador, o para una matriz de objetos, se invoca al método **operator delete** de dicha clase. En este caso, el prototipo del método tiene que coincidir con alguno de los siguientes:

```
void C::operator delete( void * );
void C::operator delete[]( void * );
void C::operator delete( void *, size_t );
void C::operator delete[]( void *, size_t );
```

Un método **operator delete** definido en una clase es un método estático (aun cuando no se declare explícitamente **static**, por lo tanto, no puede ser virtual), debe tener como tipo del resultado **void** y un primer argumento de tipo **void ***. También puede añadirse un segundo argumento de tipo **size_t** (tipo definido en el fichero *cstdlib*), el cual será iniciado por el compilador con el tamaño en bytes del objeto direccionado por el primer argumento. La razón por la que tiene que ser estático es porque se invoca después del destructor, por lo tanto, no opera sobre un objeto de la clase; en lugar de ello, actúa sobre el espacio de memoria en el que el destructor acaba de destruir un objeto. Esto quiere decir que hay un grado de separación entre la liberación y la limpieza que redundan en programas más claros.

Cuando se ejecuta **delete**, primero se busca una definición para este operador en la clase del objeto; si no se encuentra, entonces se busca en sus clases base, y por último, si no se encuentra, se ejecuta el operador **delete** global.

Como ejemplo, continuemos con la clase *CVector* y añadamos a la misma las sobrecargas del operador **delete** mostradas a continuación:

```
void CVector::operator delete(void *p, size_t tam)
{
    if (p) fill(static_cast<char *>(p), static_cast<char *>(p)+tam, 0);
    free(p);
}

void CVector::operator delete[](void *p, size_t tam)
{
    if (p) fill(static_cast<char *>(p), static_cast<char *>(p)+tam, 0);
    free(p);
}
```

La primera sobrecarga del operador **delete** será invocada cada vez que se elimine un objeto *CVector* creado por **new** y la segunda, cuando se elimine una matriz dinámica de objetos *CVector*. Para probar ambas sobrecargas, vamos a añadir a la función **main** el código que a continuación se muestra sombreado:

```
void fnVisualizar(CVector&);

int main()
{
    const int n = 3;
    CVector *pvector1 = new CVector;
    CVector *pvector2 = new CVector[n];

    fnVisualizar(*pvector1);
    for (int i = 0; i < n; i++)
        fnVisualizar(pvector2[i]);

    delete pvector1;
    delete [] pvector2;
}
```

En este ejemplo, cuando se destruye el objeto apuntado por *pvector1*, primero se invoca al destructor *~CVector()* para destruir el objeto, esto es, se libera el bloque de memoria que el constructor reservó para los *nElementos* de la matriz, y después se invoca al método *operator delete(void *, size_t)* para liberar el bloque de memoria sobre el que se construyó el objeto.

Cuando se destruye la matriz de objetos *pvector2*, primero se invoca al destructor *~CVector()* una vez por cada objeto de la matriz y después se invoca al método *operator delete[](void *, size_t)* para liberar el bloque de memoria sobre el que se construyeron los objetos elementos de la matriz. En ambos casos, la memoria liberada es puesta a cero.

EJERCICIOS RESUELTOS

1. ¿Qué es un número complejo? Un número complejo está compuesto por dos números reales y se representa de la forma $a+bi$; a recibe el nombre de componente real y b el de componente imaginario. Si $b = 0$, se obtiene el número real a , lo que quiere decir que los números reales son un caso particular de los números complejos.

Los números complejos cubren un campo que no tiene sentido en el campo de los números reales. Por ejemplo, no existe ningún número real que sea igual a $\sqrt{-9}$. Tampoco tienen sentido las expresiones $(-2)^{3/2}$ o $\log(-2)$. Para resolver este tipo de expresiones se definió la unidad imaginaria $\sqrt{-1}$, que se representa por i . De este modo podemos escribir que:

$$2 + \sqrt{-9} = 2 + 3\sqrt{-1} = 2 + 3i, \text{ que se representa como } (2, 3).$$

Puesto que un número complejo (a, b) es un par ordenado de números reales, puede representarse geoméricamente mediante un punto en el plano; dicho de otra forma, mediante un vector. De aquí se deduce que: $a+bi$, número complejo en forma binómica, es equivalente a $m(\cos \alpha + i \operatorname{sen} \alpha)$, número complejo en forma polar, lo que indica que $a = m \cos \alpha$ y que $b = m \operatorname{sen} \alpha$.

El número positivo $m = \sqrt{a^2 + b^2}$ se denomina *módulo* o *valor absoluto* y el ángulo $\alpha = \operatorname{arc} \operatorname{tg}(b/a)$ recibe el nombre de *argumento*.

Operaciones aritméticas:

Suma: $(a, b) + (c, d) = (a+c, b+d)$

Resta: $(a, b) - (c, d) = (a-c, b-d)$

Multiplicación: $(a, b) * (c, d) = (ac-bd, ad+bc)$

División: $(a, b) / (c, d) = ((ac+bd)/(c^2+d^2), (bc-ad)/(c^2+d^2))$

Estas operaciones y otras quedan perfectamente expuestas en el programa que se muestra a continuación. Las comparaciones entre complejos están referidas a sus módulos.

Según la definición dada, podemos representar un complejo como un objeto que tenga dos atributos: uno para representar la parte real y otro para representar la parte imaginaria.

```
class CComplejo
{
```

```
private:
    double real; // parte real
    double imag; // parte imaginaria
    // ...
};
```

Además, la clase *CComplejo* proveerá la funcionalidad necesaria para trabajar con números complejos. La declaración de la clase y de los métodos **inline** se escriben en el fichero *complejo.h* y el resto de las definiciones, en el fichero *complejo.cpp*.

La funcionalidad de esta clase está soportada por dos atributos, *real* e *imag*, que se corresponden con la parte real e imaginaria, respectivamente, del número complejo y varios conjuntos de funciones. Estas funciones se pueden clasificar de la forma siguiente:

- Un constructor. El complejo construido por omisión es el (0, 0). La clase *CComplejo* no necesita destructor, ya que no hay memoria alguna que liberar cuando el objeto es destruido al salir fuera de su ámbito.
- Paso de forma polar a binómica.
- Operaciones aritméticas.
- Comparación de complejos. La igualdad y la desigualdad la realizaremos en módulo y argumento. El resto de las comparaciones tienen sentido cuando sólo se comparan los módulos.
- Operaciones trigonométricas.
- Operaciones logaritmo natural, exponencial, potencia y raíz cuadrada.
- Operaciones de entrada/salida.
- Manipulación de errores.
- Obtención de valores.
- Complejo conjugado, negativo y opuesto.
- Operaciones de asignación.

La declaración de la clase y de los métodos **inline** se escriben en el fichero *complejo.h* que se muestra a continuación.

```
// complejo.h - Declaración de la clase CComplejo
#ifdef !defined(_COMPLEJO_H_)
#define _COMPLEJO_H_

#include <cmath>
```

```

#include <iostream>
using namespace std;

////////////////////////////////////
// Clase para operar con números complejos
class CComplejo
{
private:
    double real; // parte real
    double imag; // parte imaginaria
protected:
    static void error(char *);
public:
    // Constructores
    // CComplejo(const CComplejo&)
    // es creado por omisión por el compilador
    CComplejo(const double r = 0, const double i = 0)
        : real(r), imag(i) {}

    // Paso de forma polar a binómica: m(cos alfa + isen alfa)=a+bi
    friend CComplejo po_bi(const double, const double);

    // Operaciones aritméticas con complejos
    friend CComplejo operator+(const CComplejo&, const CComplejo&);
    friend CComplejo operator-(const CComplejo&, const CComplejo&);
    friend CComplejo operator*(const CComplejo&, const CComplejo&);
    friend CComplejo operator/(const CComplejo&, const CComplejo&);

    // Comparación de complejos
    friend bool operator==(const CComplejo&, const CComplejo&);
    friend bool operator!=(const CComplejo&, const CComplejo&);
    friend bool operator< (const CComplejo&, const CComplejo&);
    friend bool operator<=(const CComplejo&, const CComplejo&);
    friend bool operator> (const CComplejo&, const CComplejo&);
    friend bool operator>=(const CComplejo&, const CComplejo&);

    // Operaciones trigonométricas con complejos
    friend CComplejo cos(const CComplejo&);
    friend CComplejo sin(const CComplejo&);
    friend CComplejo tan(const CComplejo&);
    friend CComplejo cosh(const CComplejo&);
    friend CComplejo sinh(const CComplejo&);
    friend CComplejo tanh(const CComplejo&);

    // Operaciones logaritmo, exponencial, potencia y raíz cuadrada
    friend CComplejo exp(const CComplejo&);
    friend CComplejo log(const CComplejo&);
    friend CComplejo pow(const CComplejo&, const CComplejo&);
    friend CComplejo sqrt(const CComplejo& c);

    // Operaciones de entrada/salida

```

```

friend istream& operator>>(istream&, CComplejo&);
friend ostream& operator<<(ostream&, const CComplejo&);

// Obtención de valores
double ParteReal() const { return real; }
double ParteImag() const { return imag; }
double mod() const
    { return sqrt(real*real + imag*imag); }
double arg() const { return atan2(imag, real); }
double norm() const { return real*real + imag*imag; }

// Operaciones varias
CComplejo conjugado() { return CComplejo(real, -imag); }
CComplejo negativo() { return CComplejo(-real, imag); }

// Menos unario. Complejos opuestos
CComplejo operator-() { return CComplejo(-real, -imag); }

// Asignación de complejos
// operator=() lo crea el compilador por defecto
CComplejo operator+=(const CComplejo& c)
    { return *this = *this + c; }
CComplejo operator-=(const CComplejo& c)
    { return *this = *this - c; }
CComplejo operator*=(const CComplejo& c)
    { return *this = *this * c; }
CComplejo operator/=(const CComplejo& c)
    { return *this = *this / c; }
};

#endif // _COMPLEJO_H_
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

El constructor tiene dos argumentos por omisión, lo cual hace posible operar con complejos y con reales.

```

CComplejo(const double r = 0, const double i = 0)
    : real(r), imag(i) {}

```

Un constructor de esta forma evita tener que definir un constructor para convertir un **double** a un complejo.

Observe que solamente hay dos atributos privados, *real* e *imag*, que son un par de valores de tipo **double** representando la parte real e imaginaria del número complejo. También en la parte privada se incluye un método **static**, *error*, que será invocado cuando se necesite visualizar un mensaje de error durante un proceso con complejos.

El método *error* tiene un argumento que referencia a una cadena de caracteres, la correspondiente al mensaje que deseamos visualizar.

En general, los parámetros de las funciones se han definido de la forma:

```
const CComplejo& c
```

El hecho de que el parámetro de la función sea una referencia es simplemente por una cuestión de eficiencia; esto es, de esta forma se evita una llamada al constructor copia para copiar el objeto pasado y una llamada al destructor para eliminar el objeto local a la función (la copia) cuando ésta finalice. El declarar el objeto constante impide que el argumento pasado se modifique, característica implícita cuando el argumento se pasa por valor. Por ejemplo:

```
bool operator<(const CComplejo& x, const CComplejo& y)
{
    return x.mod() < y.mod();
}
```

Observe también que cuando los parámetros de una función se definen como constantes (en el ejemplo, *x* e *y* son referencias a objetos constantes) y a través de ellos se invoca a un método (en nuestro caso, al método *mod*), este método tiene que declararse y definirse como constante. Por ejemplo:

```
double mod() const
{
    return sqrt(real*real + imag*imag);
}
```

Si declaramos explícitamente un objeto de una determinada clase constante, es un error que el método invocado cuando se envía un mensaje a este objeto no sea también constante.

Muchas de las funciones se han declarado **friend** de la clase *CComplejo*. La razón, en cuanto a los operadores binarios se refiere, es que una operación binaria con un operando de un tipo predefinido y otro de la clase *CComplejo* tiene que cumplir la propiedad conmutativa. Vea la clase *CRacional* en este mismo capítulo. Para el resto, la razón ha sido respetar el formato de llamada al que estamos acostumbrados; por ejemplo, $\tan(b)$, en lugar de $b.\tan()$.

Recuerde que las funciones **friend** las puede declarar en cualquier parte de la clase ya que como no son métodos de la misma, no se ven afectadas por las palabras clave **public**, **protected** o **private**.

Las definiciones de las funciones que no fueron definidas en el fichero *complejo.h* se definen en el fichero *complejo.cpp* que se muestra a continuación.

```

/* complejo.cpp - Definición de la clase CComplejo
 */
#include <iostream>
#include <cmath>
#include "complejo.h"
using namespace std;

////////////////////////////////////
// Mensajes de error
static char *MensajeError[] = {
    "división por cero",
    "log(0)",
    "en pow(z, e), z = 0" };

// Manipulación de un error
void CComplejo::error(char *mensaje)
{
    cout << "\aerror: " << mensaje << endl;
    exit(1);
}

// Paso de forma polar a binómica: m(cos alfa + isen alfa) = a+bi
CComplejo po_bi(const double mod, const double alfa)
{
    return CComplejo(mod * cos(alfa), mod * sin(alfa));
}

// Operaciones aritméticas con complejos
CComplejo operator+(const CComplejo& x, const CComplejo& y)
{
    return CComplejo(x.real + y.real, x.imag + y.imag);
}

CComplejo operator-(const CComplejo& x, const CComplejo& y)
{
    return CComplejo(x.real - y.real, x.imag - y.imag);
}

CComplejo operator*(const CComplejo& x, const CComplejo& y)
{
    return CComplejo(x.real * y.real - x.imag * y.imag,
                    x.real * y.imag + x.imag * y.real);
}

CComplejo operator/(const CComplejo& x, const CComplejo& y)
{
    double r = 0, i = 0, divisor = y.norm();

```

```
    if (divisor != 0)
    {
        r = (x.real * y.real + x.imag * y.imag) / divisor;
        i = (x.imag * y.real - x.real * y.imag) / divisor;
    }
    else
        CComplejo::error(MensajeError[0]);
    return CComplejo(r, i);
}

// Comparación de complejos
bool operator==(const CComplejo& x, const CComplejo& y)
{
    return (x.real == y.real) && (x.imag == y.imag);
}

bool operator!=(const CComplejo& x, const CComplejo& y)
{
    return !(x == y);
}

// Para el resto de las comparaciones, comparamos módulos
bool operator<(const CComplejo& x, const CComplejo& y)
{
    return x.mod() < y.mod();
}

bool operator<=(const CComplejo& x, const CComplejo& y)
{
    return x.mod() <= y.mod();
}

bool operator>(const CComplejo& x, const CComplejo& y)
{
    return x.mod() > y.mod();
}

bool operator>=(const CComplejo& x, const CComplejo& y)
{
    return x.mod() >= y.mod();
}

// Operaciones trigonométricas con complejos
CComplejo cos(const CComplejo& c)
{
    return CComplejo(cos(c.real) * cosh(c.imag),
                    -sin(c.real) * sinh(c.imag));
}

CComplejo sin(const CComplejo& c)
```

```
{
    return CComplejo(sin(c.real) * cosh(c.imag),
                    cos(c.real) * sinh(c.imag));
}

CComplejo tan(const CComplejo& c)
{
    return sin(c) / cos(c);
}

CComplejo cosh(const CComplejo& c)
{
    return CComplejo(cosh(c.real) * cos(c.imag),
                    sinh(c.real) * sin(c.imag));
}

CComplejo sinh(const CComplejo& c)
{
    return CComplejo(sinh(c.real) * cos(c.imag),
                    cosh(c.real) * sin(c.imag));
}

CComplejo tanh(const CComplejo& c)
{
    return sinh(c) / cosh(c);
}

// Operaciones logaritmicas y exponenciales
CComplejo exp(const CComplejo& c)
{
    double m = exp(c.real);
    return CComplejo(m * cos(c.imag), m * sin(c.imag));
}

CComplejo log(const CComplejo& c)
{
    double m = c.mod();
    if (m == 0) CComplejo::error(MensajeError[1]);
    return CComplejo(log(m), c.arg());
}

// Potencia
CComplejo pow(const CComplejo& c, const CComplejo& e)
{
    if (e.real == 0 && e.imag == 0)
        return CComplejo(1, 0);
    else
        if (c.real == 0 && c.imag == 0)
            CComplejo::error(MensajeError[2]);
        return exp(log(c) * e);
}
```



```

// Raíz cuadrada
CComplejo sqrt(const CComplejo& c)
{
    return pow(c, CComplejo(0.5, 0.0));
}

// Visualizar un complejo
ostream& operator<<(ostream& os, const CComplejo& c)
{
    return os << "(" << c.ParteReal() << ", " << c.ParteImag() << ")";
}

// Leer un complejo
istream& operator>>(istream& is, CComplejo& c)
{
    double re = 0, im = 0;
    char car = '\0';

    cout << "(real, imag): ";
    is >> car;
    if (car == '(')
    {
        is >> re >> car;
        if (car == ',') is >> im >> car;
        if (car != ')') is.clear(ios::badbit); // activar flag error
    }
    else
    {
        is.putback(car); // volver el carácter leído al buffer
        is >> re;
    }
    if (is) c = CComplejo(re, im);
    return is;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

A continuación se indican algunos ejemplos de operaciones con números complejos.

```

// test.cpp - Operaciones con números complejos
#include <iostream>
#include "complejo.h"
using namespace std;

int main()
{
    CComplejo a(3.5, -0.7), b(2.0, 1.5), c(-1, 1), d;
    double mod = a.mod();
    double alfa = a.arg();
}

```

```
a = -c;
a += b;
if (a != CComplejo(0, 0)) c = b / a;
d = po_bi(mod, alfa);
d = tan(b);
d = pow(a, c);
cout << "d = " << d << endl;
cin >> d;
d += 3;
cout << "d = " << d << endl;
a = log(CComplejo(0, 0));
}
```

2. La sobrecarga de operadores es muy utilizada con clases que implementan particularmente números de tipos no predefinidos; por ejemplo, números racionales y números complejos. No obstante, se puede aplicar también a clases que den lugar a objetos no numéricos; por ejemplo, cadenas de caracteres. El siguiente ejemplo implementa una clase *CCadena* para manipular cadenas de caracteres.

La clase *CCadena* proveerá la funcionalidad necesaria para trabajar con cadenas de caracteres. La declaración de la clase y de los métodos **inline** se escriben en el fichero *cadena.h* y el resto de las definiciones, en el fichero *cadena.cpp*.

La funcionalidad de esta clase está soportada por dos atributos, *pmCad* y *nlong*, que se corresponden, respectivamente, con un puntero a la cadena de caracteres y con la longitud de la misma, y varios conjuntos de funciones. Estas funciones se pueden clasificar de la forma siguiente:

- Un constructor que tiene un parámetro de tipo **const char *** que toma el valor 0 por omisión. Si no se le pasa ningún argumento, construye una cadena nula (cadena de longitud 0, no un puntero nulo). Si se le pasa como argumento una cadena de caracteres, la convierte a un objeto *CCadena*. También se ha dotado a la clase de un constructor copia que crea un objeto, copia de otro existente, y de otro constructor para construir cadenas de *n* caracteres iguales a otro carácter dado.
- Un destructor. La clase *CCadena* necesita un destructor para liberar la memoria ocupada por la cadena de caracteres apuntada por el atributo *pmCad* cuando el objeto correspondiente sale fuera de su ámbito.
- Métodos para enlazar (concatenar) cadenas de caracteres (+ y +=).
- Comparación de cadenas de caracteres. Por similitud con las funciones C, se han considerado solamente los operadores ==, <, > y además !=.

- Operaciones de entrada/salida (>> y <<).
- Operadores de asignación = y +=. Permiten asignar el contenido de un objeto o de una cadena de caracteres a otro objeto.
- Operador de indexación. Permite acceder a un carácter individual en una cadena (por ejemplo, *car = cadena[7]*).
- Manipulación de errores.
- Método para obtener la longitud de una cadena.

La declaración de la clase y de los métodos **inline** se escriben en el fichero *cadena.h* que se muestra a continuación.

```
// cadena.h - Declaración de la clase CCadena
#ifndef _CADENA_H_
#define _CADENA_H_

#include <iostream>
using namespace std;

////////////////////////////////////
// Clase para operar con cadenas de caracteres
class CCadena
{
private:
    char *pmCad; // puntero a la cadena de caracteres
    size_t nlong; // longitud de la cadena
protected:
    static void error(char *);
public:
    CCadena(const char * = 0); // constructor
    CCadena(const CCadena&); // constructor copia
    CCadena(char, int); // constructor
    ~CCadena(); // destructor

    // Concatenar cadenas de caracteres
    friend CCadena operator+(const CCadena&, const CCadena&);

    // Comparación de cadenas
    friend bool operator==(const CCadena&, const CCadena&);
    friend bool operator!=(const CCadena&, const CCadena&);
    friend bool operator< (const CCadena&, const CCadena&);
    friend bool operator> (const CCadena&, const CCadena&);

    // Operaciones de entrada/salida
    friend istream& operator>>(istream&, CCadena&);
    friend ostream& operator<<(ostream&, const CCadena&);

    // Asignación, concatenación e indexación
```

```

    CCadena& operator=(const CCadena&); // asignación objeto
    CCadena& operator=(const char *); // asignación cadena
    CCadena operator+=(const CCadena&); // suma mas asignación
    char& operator[](unsigned int); // indexación
    size_t ObtenerLong() const { return nlong; }
};

#endif // _CADENA_H_
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

Una operación común a los constructores y a los métodos de asignación y concatenación es utilizar el operador **new** para asignar el espacio de memoria necesario para contener la cadena. Como resultado, cada objeto *CCadena* está formado por dos bloques de memoria, uno donde se construye el objeto (estructura de datos formada por *pmCad* y *nlong*) y otro que almacena la cadena de caracteres apuntada por *pmCad*.

La clase *CCadena* requiere un destructor que utilice el operador **delete** para liberar la memoria asignada por **new**. De lo contrario, el destructor por omisión liberaría la memoria ocupada por el objeto, pero no la ocupada por la cadena de caracteres referenciada por el mismo.

Sabemos que cuando no se define un operador de asignación, el compilador añade uno por omisión. La definición de éste sería de la forma:

```

CCadena& CCadena::operator=(const CCadena& x)
{
    pmCad = x.pmCad;
    nlong = x.nlong;
    return *this;
}

```

Este método realiza la asignación de un objeto a otro, miembro a miembro. Como consecuencia, los dos objetos referenciarán una misma cadena de caracteres. Esto significa que cualquier modificación en uno de los objetos afecta al otro, y quizás no sea esto lo que deseemos. Además del problema anterior, se presentará otro más serio cuando se salga fuera del ámbito de uno de los objetos, ya que el destructor de la clase eliminaría la cadena común. Por esta razón se ha implementado un operador de asignación que crea una cadena única para cada objeto.

Igual que sucede con los operadores << y >>, el método **operator=** devuelve un valor de tipo *CCadena&*. Esto permite utilizar el operador = de forma encadenada. Por ejemplo:

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

Las definiciones de los métodos de la clase *CCadena*, así como de las funciones **friend**, se localizan en el fichero *cadena.cpp* que se presenta a continuación.

```

/* cadena.cpp - Definición de la clase CCadena
 */
#include <iostream>
#include "cadena.h"
using namespace std;

////////////////////////////////////
// Mensajes de error
static char *MensajeError[] = { "insuficiente memoria" };

// Manipulación del error
void CCadena::error(char *mensaje)
{
    cerr << "\aerror: " << mensaje << endl;
    exit(1);
}

// Constructores
// Constructor para: CCadena c y CCadena c("cadena")
CCadena::CCadena(const char *pcad)
{
    if (pcad == 0) // construir una cadena nula
        nlong = 0;
    else
        nlong = strlen(pcad);

    pmCad = new (nothrow) char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    if (pcad == 0) // construir una cadena nula
        *pmCad = 0;
    else
        strcpy(pmCad, pcad);
}

// Constructor copia
CCadena::CCadena(const CCadena& x)
{
    nlong = x.nlong;
    pmCad = new (nothrow) char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    strcpy(pmCad, x.pmCad);
}

// Construye una cadena de n caracteres igual a car
CCadena::CCadena(char car, int n)
{
    nlong = n;

```

```
    pmCad = new (nothrow) char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    memset(pmCad, car, nlong);
    pmCad[nlong] = '\0';
}

// Destructor. Liberar memoria
CCadena::~CCadena()
{
    delete [] pmCad;
}

// Operadores de asignación
CCadena& CCadena::operator=(const CCadena& x)
{
    nlong = x.nlong;
    delete [] pmCad;
    pmCad = new (nothrow) char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    strcpy(pmCad, x.pmCad);
    return *this;
}

CCadena& CCadena::operator=(const char *c)
{
    nlong = strlen(c);
    delete [] pmCad;
    pmCad = new (nothrow) char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    strcpy(pmCad, c);
    return *this;
}

// Operador de indexación
char& CCadena::operator[](unsigned int ind)
{
    if (ind < 0 || nlong < ind)
        cerr << "error: índice fuera de rango\n";
    return pmCad[ind];
}

// Concatenar cadenas de caracteres

CCadena operator+(const CCadena& x, const CCadena& y)
{
    CCadena CadTemp;
    CadTemp.nlong = x.nlong + y.nlong;
    // Liberar la cadena nula asignada por el constructor
    delete [] CadTemp.pmCad;
    CadTemp.pmCad = new (nothrow) char[CadTemp.nlong + 1];
    if (!CadTemp.pmCad) CCadena::error(MensajeError[0]);
```

```

    strcpy(CadTemp.pmCad, x.pmCad);
    strcat(CadTemp.pmCad, y.pmCad);
    return CadTemp;
}

CCadena CCadena::operator+=(const CCadena& x)
{
    return *this = *this + x;
}

// Comparación de cadenas
bool operator==(const CCadena& x, const CCadena& y)
{
    return strcmp(x.pmCad, y.pmCad) == 0;
}

bool operator!=(const CCadena& x, const CCadena& y)
{
    return !(x == y);
}

bool operator<(const CCadena& x, const CCadena& y)
{
    return strcmp(x.pmCad, y.pmCad) < 0;
}

bool operator>(const CCadena& x, const CCadena& y)
{
    return strcmp(x.pmCad, y.pmCad) > 0;
}

// Operaciones de entrada/salida
ostream& operator<<(ostream& os, const CCadena& x)
{
    return os << x.pmCad;
}

istream& operator>>(istream& is, CCadena& x)
{
    char cadena[256], c;
    is.get (cadena, 256, '\n');

    if (is.get(c) && c != '\n')
        cerr << "Cadena demasiado larga; se ha truncado\n";
    x = cadena; // llama al operador =
    return is;
}

```

A continuación se indican algunos ejemplos de operaciones con cadenas de caracteres.

```

// test.cpp - Cadenas de caracteres
#include <iostream>
#include "cadena.h"
using namespace std;

int main()
{
    CCadena a[10], b; // llama al constructor
    char sCadena[] = "abcdef";

    a[0] = "xxx";
    a[2] = a[1] = a[0];
    a[3] = sCadena;
    CCadena x = a[3];
    a[4] = a[0] + sCadena;
    a[5] = "yyy" + CCadena("zzz");
    b = CCadena('/', 10);
    cout << b[4] << '\n';
    cout << "introduce cadena: "; cin >> a[6];
    (a[6] > b) ? cout << a[6] << '\n' : cout << b << '\n';
    if (a[0] == "xxx" && "yyy" > a[3])
        a[7] += "zzz";
    if ("xxx" != a[1] || sCadena < a[2])
        a[8] = a[8] + "fin";

    for(int i = 0; i < 10; i++)
        cout << a[i] << " " << a[i].ObtenerLong() << '\n';
}

```

Observe que el constructor no sólo permite construir un objeto *CCadena*, sino que además permite convertir una cadena de caracteres a un objeto *CCadena*, ya que su argumento es un puntero a una cadena. El resultado final es que se pueden realizar operaciones como la siguiente:

```
a[4] = a[0] + sCadena;
```

La ejecución de la sentencia anterior ocurre de la forma siguiente:

1. Se llama al constructor *CCadena(const char *pcad)* y se convierte la cadena *sCadena* en un objeto *CCadena* temporal que denominaremos *t1*.
2. Se llama a la función **operator+** para realizar la concatenación de las dos cadenas. Como esta función define el objeto *CadTemp*, se invoca de nuevo al constructor para construir este objeto. Se efectúa la suma de *a[0]* con *t1*, dejando el resultado en *CadTemp*.

3. Cuando **operator+** ejecuta su orden **return** *CadTemp*, invoca al constructor copia para construir un objeto copia de *CadTemp* que denominaremos *t2*. Después de ejecutarse **return**, se invoca al destructor para destruir *CadTemp*.
4. Se llama al método **operator=** para asignar el resultado al objeto *a[4]*. Este método copia *t2* en *a[4]*.
5. Se invoca al destructor, dos veces, para destruir *t2* y *t1*, en este orden.

Así mismo, observe que cuando se construye un objeto utilizando los parámetros por omisión, el constructor asigna al objeto una cadena de longitud 0:

```
CCadena::CCadena(const char *pcad)
{
    if (pcad == 0) // construir una cadena nula
        nlong = 0;
    else
        nlong = strlen(pcad);
    pmCad = new char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    if (pcad == 0) // construir una cadena nula
        *pmCad = 0;
    else
        strcpy(pmCad, pcad);
}
```

Si al construir un objeto utilizando los parámetros por omisión simplemente asignáramos los valores por omisión a los atributos privados,

```
CCadena::CCadena(const char *pcad)
{
    if (pcad == 0)
    {
        pmCad = 0;
        nlong = 0;
        return;
    }
    nlong = strlen(pcad);
    pmCad = new char[nlong + 1];
    if (!pmCad) CCadena::error(MensajeError[0]);
    strcpy(pmCad, pcad);
}
```

operaciones como

```
a[7] += "zzz"; // invoca a la función operator+
```

darían lugar a un error, ya que al ejecutar **operator+** la sentencia

```
strcpy(CadTemp.pmCad, x.pmCad);
```

se encontraría con que *CadTemp.pmCad* no apunta a una cadena válida, sino que es un puntero nulo.

EJERCICIOS PROPUESTOS

1. Partiendo de las clases *CEstudios*, *CAumno*, *CAsignatura*, *CConvocatoria* y *CFecha* construidas en el capítulo anterior (apartado *Ejercicios propuestos*), sobrecargar el operador de inserción en las clases *CEstudios*, *CAumno* y *CAsignatura* para que se puedan realizar las operaciones siguientes:

```
CEstudios estudios_E;
CAumno alumno_A, alumno_B;
CAsignatura asignatura_S, asignatura_T;
CConvocatoria convocatoria_C, convocatoria_D;
// ...
asignatura_S << convocatoria_C << convocatoria_D;
estudios_E << alumno_A << alumno_B;
```

Añadir los *operadores de conversión* adecuados para que se pueda ejecutar el siguiente código:

```
void mostrar_expediente(CEstudios& v, int dni)
{
    if (!v)
    {
        cerr << "No hay alumnos matriculados\n";
        return;
    }
    // ...
}

bool CAumno::estaEnActa(int id, int& i)
{
    // ...
    if (asig) // si la asignatura está aprobada ...
        return false;
}
```

2. Partiendo de las clases *CTermino* y *CPolinomio* construidas en el capítulo anterior (apartado *Ejercicios propuestos*) y almacenadas en los ficheros *polinomio.h*, *polinomio.cpp*, *termino.h* y *termino.cpp*,
 1. Sustituya el método *Sumar* por la sobrecarga del operador +.
 2. ¿Es necesario un constructor copia? ¿Por qué? En caso afirmativo escríbalo.

3. ¿Es necesario sobrecargar el operador de asignación? ¿Por qué? En caso afirmativo escríbalo.
4. Sustituya el método *CPolinomio::VisualizarPol* por la sobrecarga del operador de inserción.
5. Qué métodos de las clases *CTermino* y *CPolinomio* se invocan y en qué orden cuando se ejecuta la siguiente sentencia:

```
PolinomioR = PolinomioA + PolinomioB;
```
6. Sobrecargue el operador adecuado para que se evalúen expresiones de la forma:

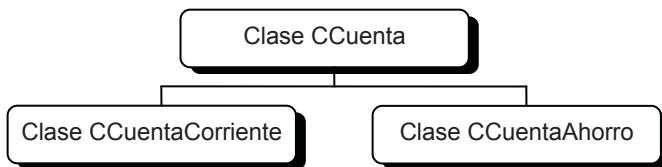
```
cout << PolinomioR(5); // valor del polinomio para x = 5
```
7. Escriba el operador de conversión adecuado para que se evalúen expresiones de la forma:

```
double v = PolinomioR; // valor del polinomio para x = 1
```
8. Sobrecargue los operadores `==`, `<` y `>` en la clase *CTermino* para saber con respecto a dos términos cuál es el de exponente mayor.
9. Reescriba el método que suma dos polinomios para que utilice los operadores de relación de la clase *CTermino*.
10. Si los operadores de relación de la clase *CTermino* los declara como métodos privados, ¿qué ocurre?
11. El problema que se le ha presentado en el apartado anterior, ¿podría solucionarlo declarando la clase *CPolinomio* amiga de *CTermino*? ¿Por qué?

CLASES DERIVADAS

Las características fundamentales de la POO son *abstracción*, *encapsulamiento*, *herencia* y *polimorfismo*. Hasta ahora sólo hemos abordado la *abstracción* y la *encapsulación*. Aunque todas ellas son fundamentales, hay una que destaca: la *herencia*. La *herencia* provee el mecanismo más simple para especificar una forma alternativa de acceso a una clase existente, o bien para definir una nueva clase que añade nuevas características a una clase existente. Esta nueva clase se denomina *subclase* o *clase derivada* y la clase existente, *superclase* o *clase base*.

Con la herencia todas las clases están clasificadas en una jerarquía estricta. Cada clase tiene su superclase (la clase superior en la jerarquía), y cada clase puede tener una o más subclases (las clases inferiores en la jerarquía). Las clases que están en la parte inferior en la jerarquía se dice que *heredan* de las clases que están en la parte superior en la jerarquía. Por ejemplo, la figura siguiente indica que las clases *CCuentaCorriente* y *CCuentaAhorro* heredan de la clase *CCuenta*.



Una jerarquía de clases muestra cómo los objetos se derivan de otros objetos más simples heredando su comportamiento. Los usuarios de C++ y de otros lenguajes de programación orientada a objetos están acostumbrados a ver jerarquías de clases para describir la *herencia*.

CLASES DERIVADAS Y HERENCIA

Vuelva a echar una ojeada a la figura mostrada al principio de este capítulo. Se trata de una jerarquía de clases que puede ser analizada desde dos puntos de vista:

1. Cuando en un principio se abordó el diseño de una aplicación para administrar las cuentas de una entidad bancaria, fue suficiente con las capacidades proporcionadas por la clase *CCuenta*. Posteriormente, la evolución de los mercados bancarios sugirió nuevas modalidades de cuentas. La mejor solución para adaptar la aplicación a esas nuevas exigencias fue definir una clase derivada de *CCuenta* para cada nueva modalidad, puesto que el mecanismo de herencia ponía a disposición de éstas todo el código de su clase base, al que sólo era necesario añadir las nuevas especificaciones. Resulta por lo tanto evidente que la *herencia* es una forma sencilla de *reutilizar el código*.
2. Cuando se abordó el diseño de una aplicación para administrar las cuentas de una entidad bancaria, la solución fue diseñar una clase especializada para cada una de las cuentas y agrupar el código común en una clase base de éstas.

En los dos casos planteados, la herencia es la solución para reutilizar código perteneciente a otras clases. Para ilustrar el mecanismo de herencia vamos a implementar la jerarquía de clases de la figura anterior. La idea es diseñar una aplicación para administrar las cuentas corrientes y de ahorro de los clientes de una entidad bancaria. Como ambas cuentas tienen bastantes cosas en común, hemos decidido agrupar éstas en una clase *CCuenta* de la cual posteriormente derivaremos las cuentas específicas que vayan surgiendo. Según este planteamiento, no parece que tengamos intención de crear objetos de *CCuenta*; más bien la intención es que agrupe el código común que heredarán sus subclases, razón por la cual, más adelante, cuando estudiemos clases abstractas, la declararemos abstracta.

Pensemos entonces inicialmente en el diseño de la clase *CCuenta*. Después de un análisis acerca de los factores que intervienen en una cuenta en general, llegamos a la conclusión de que los atributos y métodos comunes a cualquier tipo de cuenta son los siguientes:

Atributo	Significado
<i>nombre</i>	Dato de tipo string que almacena el nombre del propietario de la cuenta.
<i>cuenta</i>	Dato de tipo string que almacena el número de la cuenta.
<i>saldo</i>	Dato de tipo double que almacena el saldo de la cuenta.
<i>tipoDeInteres</i>	Dato de la clase de tipo double que almacena el tipo de interés.

Método	Significado
<i>CCuenta</i>	Es el constructor de la clase. Inicia los datos <i>nombre</i> , <i>cuenta</i> , <i>saldo</i> y <i>tipoDeInteres</i> con los valores pasados como argumentos en la llamada, o con los valores especificados por omisión.
<i>asignarNombre</i>	Permite asignar el dato <i>nombre</i> . Retorna false si el nombre es una cadena vacía, y true en otro caso.
<i>obtenerNombre</i>	Retorna el dato <i>nombre</i> .
<i>asignarCuenta</i>	Permite asignar el dato <i>cuenta</i> . Retorna false si la cuenta es una cadena vacía, y true en otro caso.
<i>obtenerCuenta</i>	Retorna el dato <i>cuenta</i> .
<i>estado</i>	Retorna el saldo de la cuenta.
<i>comisiones</i>	Es un método sin parámetros que se ejecutará el día 1 de cada mes para cobrar el importe del mantenimiento de una cuenta.
<i>ingreso</i>	Es un método que tiene un parámetro <i>cantidad</i> de tipo double que añade la cantidad especificada al saldo actual de la cuenta. Retorna false si la cantidad es negativa, y true en otro caso.
<i>reintegro</i>	Es un método que tiene un parámetro <i>cantidad</i> de tipo double que resta la cantidad especificada del saldo actual de la cuenta.
<i>asignarTipoDeInteres</i>	Método que permite asignar el dato <i>tipoDeInteres</i> . Retorna false si el tipo de interés es negativo, y true en otro caso.
<i>obtenerTipoDeInteres</i>	Método que retorna el dato <i>tipoDeInteres</i> .
<i>intereses</i>	Método que calcula los intereses producidos.

El código correspondiente a esta clase se expone a continuación:

```
// cuenta.h - clase CCuenta
// Clase base de CCuentaAhorro y CCuentaCorriente
#ifdef !_CUENTA_H_
#define _CUENTA_H_
#include <string>

class CCuenta
{
    // Atributos
private:
    std::string nombre;
    std::string cuenta;
    double saldo;
    double tipoDeInteres;
    // Métodos
```

```

public:
    CCuenta(std::string nom = "sin nombre", std::string cue = "0000",
            double sal = 0.0, double tipo = 0.0);
    bool asignarNombre(std::string nom);
    std::string obtenerNombre() const;
    bool asignarCuenta(std::string cue);
    std::string obtenerCuenta() const;
    double estado() const;
    void comisiones();
    double intereses();
    bool ingreso(double cantidad);
    void reintegro(double cantidad);
    double obtenerTipoDeInteres() const;
    bool asignarTipoDeInteres(double tipo);
};

#endif // _CUENTA_H_

```

```
// cuenta.cpp - Definición de la clase CCuenta
```

```

#include <iostream>
#include "cuenta.h"
using namespace std;

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

// Clase CCuenta: clase que agrupa los datos comunes a
// cualquier tipo de cuenta bancaria.
//

```

```

CCuenta::CCuenta(string nom, string cue, double sal, double tipo) :
nombre(nom), cuenta(cue), saldo(sal), tipoDeInteres(tipo)
{
    // Verificar los datos asignados
    asignarNombre(nom);
    asignarCuenta(cue);
    if (sal < 0) saldo = 0;
    asignarTipoDeInteres(tipo);
}

```

```

bool CCuenta::asignarNombre(string nom)
{
    if (nom.length() != 0)
        nombre = nom;
    else
        cout << "Error: cadena nombre vacía\n";
    return nom.length() != 0;
}

```

```

string CCuenta::obtenerNombre() const
{
    return nombre;
}

```



```
bool CCuenta::asignarCuenta(string cue)
{
    if (cue.length() != 0)
        cuenta = cue;
    else
        cout << "Error: cuenta no válida\n";

    return cue.length() != 0;
}

string CCuenta::obtenerCuenta() const
{
    return cuenta;
}

double CCuenta::estado() const
{
    return saldo;
}

void CCuenta::comisiones()
{
    return; // sin comisiones
}

double CCuenta::intereses()
{
    return 0.0; // sin intereses
}

bool CCuenta::ingreso(double cantidad)
{
    if (cantidad >= 0)
        saldo += cantidad;
    else
        cout << "Error: cantidad negativa\n";
    return cantidad >= 0;
}

void CCuenta::reintegro(double cantidad)
{
    if (saldo - cantidad < 0)
    {
        cout << "Error: no dispone de saldo\n";
        return;
    }
    saldo -= cantidad;
}
```

```

double CCuenta::obtenerTipoDeInteres() const
{
    return tipoDeInteres;
}

bool CCuenta::asignarTipoDeInteres(double tipo)
{
    if (tipo >= 0)
        tipoDeInteres = tipo;
    else
        cout << "Error: tipo no válido\n";
    return tipo >= 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

DEFINIR UNA CLASE DERIVADA

Pensemos ahora en un tipo de cuenta específico, como es una cuenta de ahorro. Una cuenta de ahorro tiene las características aportadas por un objeto *CCuenta*, y además algunas otras; por ejemplo, un atributo que especifique el importe que hay que pagar mensualmente por el mantenimiento de la misma. Esto significa que necesitamos diseñar una nueva clase, *CCuentaAhorro*, que tenga las mismas capacidades de *CCuenta*, pero a las que hay que añadir otras que den solución a las nuevas necesidades.

Una forma de hacer esto sería definir una nueva clase *CCuentaAhorro* con los atributos y métodos de *CCuenta*, a los que añadiríamos los nuevos atributos y métodos, según muestra el diseño siguiente:

```

class CCuentaAhorro
{
    // Atributos y métodos de CCuenta
    // Nuevos atributos y métodos de CCuentaAhorro
};

```

Esta forma de proceder puede que funcione, pero no deja de ser una mala solución; además de suponer un derroche de tiempo y esfuerzo, todo el trabajo que ya estaba realizado no ha servido para nada. Aquí es donde la herencia juega un papel importante; la utilización de esta característica evitará que recurramos a soluciones como la planteada, o a otras como la siguiente:

```

class CCuentaAhorro
{
    CCuenta c; // inclusión mediante c
    // Nuevos atributos y métodos de CCuentaAhorro
};

```

Según este otro planteamiento, una cuenta de ahorro también es una cuenta; es decir, este diseño especifica que cada operación no definida en la clase *CCuentaAhorro* relativa a *CCuenta* puede ser servida por un objeto *c* de *CCuenta*. En cambio, aunque esto pueda parecer evidente, no hay nada que indique al compilador que un objeto *CCuentaAhorro* es también un objeto *CCuenta*. Y tampoco un *CCuentaAhorro ** es un *CCuenta **, con lo cual no será nada fácil, aunque no imposible, construir una lista de objetos de cuentas diferentes. El enfoque correcto es especificar de manera explícita que *CCuentaAhorro* es una extensión *CCuenta*. Y esto, ¿cómo se hace? A través de la herencia, definiendo una clase derivada de la clase existente.

El ejemplo mostrado a continuación define la clase *CCuentaAhorro* como una extensión de *CCuenta*:

```
class CCuentaAhorro : public CCuenta
{
    // CCuentaAhorro ha heredado los miembros de CCuenta
    // Escriba aquí los nuevos atributos y métodos de CCuentaAhorro
}
```

El símbolo **:** significa que se está definiendo una clase denominada *CCuentaAhorro* que es una extensión de otra denominada *CCuenta*; en C++ se dice que *CCuentaAhorro* es una clase derivada de *CCuenta*.

Control de acceso a la clase base

Según lo expuesto, una *clase derivada* es un nuevo tipo de objetos definido por el usuario que tiene la propiedad de heredar los atributos y métodos de otra clase definida previamente, denominada *clase base*. La sintaxis para definir una *clase derivada* es la siguiente:

```
class nombre_clase_d : [{private|protected|public}] nombre_clase_b_1
[, [{private|protected|public}] nombre_clase_b_2]...
{
    cuerpo de la clase derivada
};
```

Los especificadores de acceso para una clase base controlan el acceso a los miembros de la clase base y la conversión de objetos, de punteros y de referencias desde el tipo definido por la clase derivada al definido por la clase base. Las conversiones las veremos un poco más adelante en este mismo capítulo.

La palabra clave **private** se especificará cuando en una derivación se quiera hacer la clase base privada, **protected** cuando se quiera hacer protegida y **public**

cuando se quiera hacer pública. Por omisión, se asume que es privada. Por ejemplo:

```
class A {}; // clase base
class B : private A {}; // clase base privada
class C : protected A {}; // clase base protegida
class D : public A {}; // clase base pública
```

Si en una derivación la clase base es *privada*, sus miembros **public** y **protected** pasan a ser privados (**private**) en la clase derivada, por lo tanto sólo se podrán utilizar en los métodos y funciones amigas de la derivada; sus miembros **private** siguen siendo privados.

Si en una derivación la clase base es *protegida*, sus miembros **public** y **protected** pasan a ser protegidos (**protected**) en la clase derivada, por lo tanto sólo se podrán utilizar en los métodos y funciones amigas de la derivada y en los métodos y funciones amigas de las clases que a su vez se deriven de ésta; sus miembros **private** siguen siendo privados.

Si en una derivación la clase base es *pública*, sus miembros **public** siguen siendo públicos en la clase derivada sus miembros **protected** siguen siendo protegidos y sus miembros **private** siguen siendo privados (ver a continuación *El control de acceso a los miembros de las clases*).

Una clase derivada puede, a su vez, ser una clase base de otra clase, dando lugar así a una *jerarquía de clases*. Por lo tanto, una clase será una clase base directa de una clase derivada, si figura explícitamente en la definición de la clase derivada, o una clase base indirecta si está varios niveles arriba en la jerarquía de clases, y por lo tanto no figura explícitamente en el encabezado de la definición de la clase derivada.

Cuando una clase derivada lo es de una sola clase base, la herencia se denomina *herencia simple* o *derivación simple*; en cambio, cuando lo es de dos o más clases, la herencia se denomina *múltiple* o *derivación múltiple*. C++, a diferencia de otros lenguajes orientados a objetos, permite la herencia múltiple.

Control de acceso a los miembros de las clases

En el capítulo dedicado a clases se expuso que para controlar el acceso a los miembros de una clase, C++ provee las palabras clave **private** (privado), **protected** (protegido) y **public** (público). Lo allí estudiado se amplía ahora para las clases derivadas. Para evitar confusiones, la tabla siguiente resume de una forma clara qué clases, clases derivadas o funciones pueden acceder a los miembros de

otra clase, dependiendo del control de acceso especificado. Evidentemente, cuando hablamos de acceso a un miembro nos referimos al acceso directo al mismo a través de su nombre.

Puede ser accedido desde:	Un miembro que en una clase es		
	<i>privado</i>	<i>protegido</i>	<i>público</i>
Su misma clase (métodos y funciones amigas)	sí	sí	sí
Cualquier clase derivada	no	sí	sí
Cualquier otra clase no derivada (y no amiga)	no	no	sí
Cualquier función externa	no	no	sí

De lo expuesto se deduce que una clase derivada de otra clase procedente de una derivación privada no tiene acceso a ningún miembro. Aunque esta restricción puede sorprender, es así para imponer la encapsulación. De otra forma, esto es, si una clase derivada tuviera acceso a los miembros privados de su clase base, bastaría derivar una clase de cualquier otra para acceder a sus miembros privados.

Qué miembros hereda una clase derivada

Los siguientes puntos resumen las reglas a tener en cuenta cuando se define una clase derivada:

1. Una clase derivada hereda todos los miembros de su clase base, excepto los constructores, lo cual no significa que tenga acceso directo a todos los miembros. Una consecuencia inmediata de esto es que la estructura interna de datos de un objeto de una clase derivada estará formada por los atributos que ella define y por los heredados de su clase base.

Una clase derivada no tiene acceso directo a los miembros privados (**private**) de su clase base, pero sí puede acceder a los miembros públicos (**public**) y protegidos (**protected**).

2. Una clase derivada puede añadir sus propios atributos y métodos. Si el nombre de alguno de estos miembros coincide con el de un miembro heredado, éste último queda oculto para la clase derivada, lo que se traduce en que la clase derivada ya no puede acceder directamente a ese miembro. Lógicamente, lo expuesto tiene sentido siempre que nos refiramos a los miembros de la

clase base a los que la clase derivada podía acceder, según el control de acceso aplicado.

3. Los miembros heredados por una clase derivada pueden, a su vez, ser heredados por más clases derivadas de ella. A esto se le llama propagación de la herencia.

Continuando con el ejemplo, diseñemos una nueva clase *CCuentaAhorro* que tenga, además de las mismas capacidades de *CCuenta*, las siguientes:

Atributo	Significado
<i>cuotaMantenimiento</i>	Dato de tipo double que almacena la comisión que cobrará la entidad bancaria por el mantenimiento de la cuenta.

Método	Significado
<i>CCuentaAhorro</i>	Es el constructor de la clase. Inicia los atributos de la misma.
<i>asignarCuotaManten</i>	Establece la cuota de mantenimiento de la cuenta. Retorna false si la cantidad es negativa, y true en otro caso.
<i>obtenerCuotaManten</i>	Devuelve la cuota de mantenimiento de la cuenta.
<i>comisiones</i>	Método que se ejecuta los días 1 de cada mes para cobrar el importe correspondiente al mantenimiento de la cuenta.
<i>intereses</i>	Método que permite calcular el importe correspondiente a los intereses/mes producidos, los cuales serán abonados los días 1 de cada mes.

Los métodos *comisiones* e *intereses* obtienen la fecha actual a través de la funcionalidad proporcionada por la clase *CFecha* desarrollada en los capítulos anteriores. Veamos a continuación la declaración y definición de *CCuentaAhorro*:

```
// cuenta_ahorro.h - Declaración de la clase CCuentaAhorro
#ifndef _CUENTA_AHORRO_H_
#define _CUENTA_AHORRO_H_
#include "cuenta.h"

class CCuentaAhorro : public CCuenta
{
    // Atributos
private:
    double cuotaMantenimiento;

    // Métodos
public:
    CCuentaAhorro() {} // constructor sin parámetros
    bool asignarCuotaManten(double);
};
```

```

        double obtenerCuotaManten() const;
        void comisiones();
        double intereses();
};

#endif // _CUENTA_AHORRO_H_

// cuenta_ahorro.cpp - Definición de la clase CCuentaAhorro
#include <iostream>
#include "cuenta_ahorro.h"
#include "fecha.h"
using namespace std;

////////////////////////////////////
// Clase CCuentaAhorro: clase derivada de CCuenta
//
bool CCuentaAhorro::asignarCuotaManten(double cantidad)
{
    if (cantidad >= 0)
        cuotaMantenimiento = cantidad;
    else
        cout << "Error: cantidad negativa\n";

    return cantidad >= 0;
}

double CCuentaAhorro::obtenerCuotaManten() const
{
    return cuotaMantenimiento;
}

void CCuentaAhorro::comisiones()
{
    // Se aplican mensualmente por el mantenimiento de la cuenta
    int dia, mes, anyo;
    CFecha::obtenerFechaActual(dia, mes, anyo);

    if (dia == 1) reintegro(cuotaMantenimiento);
}

double CCuentaAhorro::intereses()
{
    int dia, mes, anyo;
    CFecha::obtenerFechaActual(dia, mes, anyo);

    if (dia != 1) return 0.0;
    // Acumular los intereses por mes sólo los días 1 de cada mes
    double interesesProducidos = 0.0;
    interesesProducidos = estado() * obtenerTipoDeInteres() / 1200.0;
    ingreso(interesesProducidos);
}

```

```
// Devolver el interés mensual por si fuera necesario
return interesesProducidos;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

CCuentaAhorro es una clase derivada de la clase base *CCuenta*. Obsérvese que para definir una clase derivada se añade a continuación del nombre de la misma el símbolo **:** seguido del tipo de derivación y del nombre de la clase base. En la definición de la clase derivada se describen las características adicionales que la distinguen de la clase base.

Veamos a continuación una comparativa de las clases base y derivada. La capacidad de la clase *CCuenta* está soportada por:

Atributos	Métodos
nombre	constructor <i>CCuenta</i>
cuenta	constructor copia, destructor y operador =, por omisión
saldo	asignarNombre
tipoDeInteres	obtenerNombre
	asignarCuenta
	obtenerCuenta
	estado
	comisiones
	intereses
	ingreso
	reintegro
	asignarTipoDeInteres
	obtenerTipoDeInteres

Y la capacidad de la clase *CCuentaAhorro*, derivada de *CCuenta*, está soportada por los miembros heredados de *CCuenta* (en cursiva y no tachados) más los suyos (en letra normal):

Atributos	Métodos
<i>nombre</i>	<i>constructor CCuenta</i>
<i>cuenta</i>	<i>constructor copia, destructor y operador =, por omisión</i>
<i>saldo</i>	<i>asignarNombre</i>
<i>tipoDeInteres</i>	<i>obtenerNombre</i>
	<i>asignarCuenta</i>
	<i>obtenerCuenta</i>
	<i>estado</i>
	<i>comisiones</i>

Atributos	Métodos
	<i>intereses</i>
	<i>ingreso</i>
	<i>reintegro</i>
	<i>asignarTipoDeInteres</i>
	<i>obtenerTipoDeInteres</i>
cuotaMantenimiento	constructor CCuentaAhorro
	constructor copia, destructor y operador =, por omisión
	asignarCuotaManten
	obtenerCuotaManten
	comisiones
	intereses

Obsérvese que la clase *CCuenta* define un constructor, pero aunque no se definiera, el compilador generaría uno por omisión. Esto implica que los constructores no se heredan. Por la misma razón, tampoco se heredan el constructor copia, el destructor y el operador de asignación. Por otra parte, los métodos *comisiones* e *intereses* quedan ocultos por los métodos del mismo nombre de la clase *CCuentaAhorro*. Un poco más adelante veremos que es posible referirse a un miembro oculto utilizando el operador de ámbito `::` de C++: *clase_base::miembro_oculto*.

Según el análisis anterior, mientras un posible objeto *CCuenta* contendría los datos *nombre*, *cuenta*, *saldo* y *tipoDeInteres*, un objeto *CCuentaAhorro* contiene los datos *nombre*, *cuenta*, *saldo*, *tipoDeInteres* y *cuotaMantenimiento*.

Escribamos ahora una pequeña aplicación *test.cpp* que cree un objeto *CCuentaAhorro* (la aplicación incluirá los ficheros *test.cpp*, *cuenta.cpp* y *cuenta_ahorro.cpp* y los ficheros de cabecera necesarios):

```
int main()
{
    CCuentaAhorro cuenta01;
    cuenta01.asignarNombre("Un nombre");
    cuenta01.asignarCuenta("Una cuenta");
    cuenta01.asignarTipoDeInteres(2.5);
    cuenta01.asignarCuotaManten(20);
    cuenta01.ingreso(100000);
    cuenta01.reintegro(50000);
    cuenta01.comisiones();
    // cuenta01 no puede acceder a los miembros privados, como
    // cuenta, por ejemplo.
}
```

En este ejemplo se puede observar que un “objeto”, como *cuenta01* de la clase *CCuentaAhorro*, definido en el ámbito de una función no miembro de su clase (función externa o de otra clase) puede invocar a cualquiera de los métodos públicos de *CCuentaAhorro* y de *CCuenta*, pero no tiene acceso a sus miembros privados y protegidos.

Los “métodos” de una clase derivada no tienen acceso a los miembros privados de su clase base, pero sí lo tienen a sus miembros protegidos y públicos, implícitamente a través de **this** o explícitamente a través de un objeto de su clase. Por ejemplo, el método *comisiones* de la clase *CCuentaAhorro* no puede acceder al atributo *saldo* de la clase *CCuenta* porque es privado, pero sí puede acceder a su método público *reintegro*.

```
void CCuentaAhorro::comisiones()
{
    // Se aplican mensualmente por el mantenimiento de la cuenta
    int dia, mes, anyo;
    CFecha::obtenerFechaActual(dia, mes, anyo);
    if (dia == 1) reintegro(cuotaMantenimiento);
}
```

Por lo tanto, si una clase derivada quiere acceder a los miembros privados de su clase base, debe hacerlo a través de la interfaz pública o protegida de dicha clase.

ATRIBUTOS CON EL MISMO NOMBRE

Como sabemos, una clase derivada puede acceder directamente a un atributo público o protegido de su clase base. ¿Qué sucede si definimos en la clase derivada uno de estos atributos, con el mismo nombre que tiene en la clase base? Por ejemplo, supongamos que una clase *ClaseA* define un atributo identificado por *atributo_x*, que después redefinimos en una clase derivada *ClaseB*:

```
class ClaseA
{
public:
    int atributo_x;

public:
    ClaseA(int x = 1) : atributo_x(x) {}

    int metodo_x()
    {
        return atributo_x * 10;
    }
}
```

```

        int metodo_y()
        {
            return atributo_x + 100;
        }
};

class ClaseB : public ClaseA
{
public:
    int atributo_x;

public:
    ClaseB(int x = 2) : atributo_x(x) {}

    int metodo_x()
    {
        return atributo_x * -10;
    }
};

```

La definición del atributo *atributo_x* en la clase derivada oculta la definición del atributo con el mismo nombre en la clase base. Por lo tanto, las referencias a *atributo_x* en el código del ejemplo siguiente devolverán el valor de *atributo_x* de la *ClaseB*. Si este atributo no hubiera sido definido en la clase derivada, entonces el valor devuelto sería el valor de *atributo_x* de la clase base.

```

int main()
{
    ClaseB objClaseB;

    cout << objClaseB.atributo_x << endl; // escribe 2
    cout << objClaseB.metodo_y() << endl; // escribe 101
    cout << objClaseB.metodo_x() << endl; // escribe -20
}

```

Ahora bien, ¿cómo procederíamos si el método referenciado por el *metodo_x* de la clase *ClaseB* tuviera que acceder obligatoriamente al dato *atributo_x* de la clase base? La solución es sencilla: utilizar para ese atributo nombres diferentes en la clase base y en la clase derivada. No obstante, aun habiendo utilizado el mismo nombre, tenemos una alternativa de acceso: utilizar el nombre de su clase más el operador de ámbito `::`. Por ejemplo, modifiquemos el *metodo_x* de la *ClaseB* así:

```

int metodo_x() // método de la ClaseB
{
    return ClaseA::atributo_x * -10;
}

```

Como se puede ver, podemos referirnos al dato *atributo_x* de la clase base con la expresión:

```
ClaseA::atributo_x
```

Así mismo, como ya sabemos, también podríamos referirnos al dato *atributo_x* de la clase derivada con la expresión:

```
this->atributo_x
```

En cambio, la expresión siguiente hace referencia al dato *atributo_x* de la *ClaseA*:

```
static_cast<ClaseA *>(this)->atributo_x
```

REDEFINIR MÉTODOS DE LA CLASE BASE

Cuando se invoca a un método en respuesta a un mensaje recibido por un objeto, C++ busca su definición en la clase del objeto. El método que allí se encuentre puede pertenecer a la propia clase o puede haber sido heredado de alguna de sus clases base (esto último equivale a decir que si no lo encuentra, C++ sigue buscando hacia arriba en la jerarquía de clases hasta que lo localice).

Sin embargo, puede haber ocasiones en que deseemos que un objeto de una clase derivada responda al mismo método heredado de su clase base pero con un comportamiento diferente. Esto implica redefinir en la clase derivada el método heredado de su clase base.

Redefinir un método heredado significa volverlo a escribir en la clase derivada con el mismo nombre, la misma lista de parámetros y el mismo tipo del valor retornado que tenía en la clase base; su cuerpo será adaptado a las necesidades de la clase derivada. Esto es lo que se ha hecho con el *metodo_x* del ejemplo expuesto en el apartado anterior.

Se puede observar que este método ha sido redefinido en la *ClaseB* para que realice unos cálculos diferentes a los que realizaba en la *ClaseA*.

En el método **main** del ejemplo anterior, se creó un objeto *objClaseB* y se invocó a su *metodo_y*. Como la clase del objeto, *ClaseB*, no define este método, C++ ejecuta el heredado. Así mismo, se invocó a su *metodo_x*; en este caso, existe una definición para este método, que es la que se ejecuta.

Cuando en una clase derivada se redefine un método de una clase base, se oculta el método de la clase base y todas las sobrecargas que existan del mismo en

dicha clase base. Recuerde que para que exista una sobrecarga de una función, además de darse la condición de que la lista de parámetros tiene que ser diferente, hay que realizar todas las declaraciones en el mismo ámbito. Un método de una clase derivada no está en el mismo ámbito que un método con el mismo nombre en su clase base.

Si el método se escribe en la clase derivada con distinto tipo o número de parámetros, el método de la clase base también se oculta. Por ejemplo, el *metodo_x* tal cual lo hemos redefinido en la clase derivada oculta al método del mismo nombre de la clase base. Pero si lo hubiéramos definido con distinto número de parámetros, por ejemplo con uno como se muestra a continuación, según lo explicado anteriormente, no se considera una redefinición, tampoco una sobrecarga, y oculta a los métodos con el mismo nombre de la clase base.

```
int metodo_x(int a) // método de la ClaseB
{
    return atributo_x * -a;
}
```

El control de acceso de un miembro que se redefine puede modificarse en cualquier sentido; esto es, se puede hacer que sea más o menos restrictivo que el original. El orden de los tipos de control de acceso de más a menos restrictivo es así: **private**, **protected** y **public**.

Para acceder a un método de la clase base que ha sido redefinido en la clase derivada, igual que se expuso para los atributos, tendremos que utilizar el nombre de su clase más el operador de ámbito **::**. Por ejemplo, suponga que añadimos el siguiente método a la *ClaseB*:

```
int metodo_z()// método de la ClaseB
{
    atributo_x = ClaseA::atributo_x + 3;
    return ClaseA::metodo_x() + atributo_x;
}
```

Como se puede observar, podemos referirnos al *metodo_x* de la clase base con la expresión:

```
ClaseA::metodo_x()
```

Así mismo, como ya vimos cuando se expuso **this**, también podríamos referirnos al *metodo_x* de la clase derivada así:

```
this->metodo_x()
```

CONSTRUCTORES DE CLASES DERIVADAS

Sabemos que cuando se crea un objeto de una clase se invoca a su constructor. También sabemos que los constructores de la clase base no son heredados por sus clases derivadas. En cambio, cuando se crea un objeto de una clase derivada, se invoca a su constructor, que a su vez invoca al constructor sin parámetros de la clase base, que a su vez invoca al constructor de su clase base, y así sucesivamente.

Lo anteriormente expuesto se traduce en que primero se ejecutan los constructores de las clases base de arriba a abajo en la jerarquía de clases y finalmente el de la clase derivada. Esto sucede así porque una clase derivada contiene todos los atributos de su clase base, y todos tienen que ser iniciados, razón por la que el constructor de la clase derivada tiene que llamar implícita o explícitamente al de la clase base.

Sin embargo, cuando se hayan definido constructores con parámetros tanto en las clases derivadas como en las clases base, tal vez se desee construir un objeto de la clase derivada iniciándolo con unos valores determinados. En este caso, la definición ya conocida para los constructores de una clase cualquiera se extiende ahora para permitir al constructor de la clase derivada invocar explícitamente al constructor de la clase base. Esto se hace incluyendo la llamada al constructor de la clase base en la lista de iniciadores como se indica a continuación:

```
nombre_clase_derivada(lista de parámetros) : lista de iniciadores
{
    cuerpo del constructor de la clase derivada
}
```

En la definición genérica anterior correspondiente a un constructor con parámetros de una clase derivada, se observa, por una parte, la utilización de una lista de iniciadores de la forma:

```
atributo1(valor1), ..., nombre_clase_base1(lista de parámetros), ...
```

que permitirá invocar al constructor de la clase base (o constructores si la derivación es múltiple) y por otra, el cuerpo del constructor de la clase derivada:

```
nombre_clase_derivada(lista de parámetros)
{
    cuerpo del constructor de la clase derivada
}
```

¿Cómo se ejecuta este código? Independientemente del orden establecido en la lista de iniciadores, el orden de ejecución es:

1. Constructores de las clases base en el orden especificado.
2. Se construyen los atributos del objeto de la clase derivada en el orden en el que están declarados en la clase en vez de en el orden en el que aparecen en la lista de iniciadores.
3. Cuerpo del constructor de la clase derivada.

Lo anterior se traduce en que un objeto de una clase derivada se construye de abajo hacia arriba; esto es, la pila de llamadas relativas a los constructores de las clases involucradas crece hasta llegar a la clase raíz en la jerarquía de clases; en este instante, comienza a ejecutarse el constructor de esta clase base: primero se construyen sus atributos ejecutando, cuando sea necesario, los constructores de los mismos y, después, se pasa a ejecutar el cuerpo del constructor de dicha clase base; a continuación se construyen los atributos del objeto de la clase derivada y, finalmente, se ejecuta el cuerpo del constructor de la clase derivada. Este orden se aplica recursivamente por cada constructor de cada una de las clases.

Por ejemplo, aplicando la teoría expuesta, vamos a añadir a la clase *CCuentaAhorro* un constructor con parámetros. ¿Cuántos parámetros debe tener este constructor para iniciar todos los atributos de un objeto *CCuentaAhorro*? Pues tantos como atributos heredados y propios tenga la clase; en nuestro caso un objeto *CCuentaAhorro* contiene los atributos *nombre*, *cuenta*, *saldo*, *tipoDeInteres* y *cuotaMantenimiento*. Según esto, la declaración del constructor podría ser así:

```
CCuentaAhorro(string nom = "sin nombre", string cue = "0000",
               double sal = 0.0, double tipo = 0.0, double mant = 0.0);
```

y su definición así:

```
CCuentaAhorro::CCuentaAhorro(string nom, string cue, double sal,
                              double tipo, double mant) :
CCuenta(nom, cue, sal, tipo), cuotaMantenimiento(mant)
{
    asignarCuotaManten(mant); // verificar el dato mant
}
```

Obsérvese que en la lista de iniciadores se llama primero al constructor de *CCuenta*, clase base de *CCuentaAhorro*, y después se inicia el atributo *cuotaMantenimiento*. Lógicamente, la clase *CCuenta* debe tener un constructor con cuatro parámetros del tipo de los argumentos especificados. Finalmente, desde el cuerpo del constructor se invoca al método *asignarCuotaManten* para verificar la validez del valor del atributo *cuotaMantenimiento* de *CCuentaAhorro*, puesto que iniciado ya estaba; quiere esto decir que, en este caso, podríamos haber prescindido de ejecutar la iniciación de *cuotaMantenimiento* en la lista de iniciadores, no obstante, en algunas ocasiones esta forma de proceder evitará resultados inesperados.


```
    asignarCuotaManten(mant);
}
```

En este caso, una declaración como la siguiente invocaría al constructor *CCuenta* sin argumentos, lo que supondría iniciar los atributos heredados de la clase base con los valores por omisión en vez de con los valores especificados.

```
CCuentaAhorro cuenta02("cliente02", "1111111111", 200000, 1.75, 10);
```

¿Sería correcto invocar al constructor de la clase base desde la primera línea del cuerpo del constructor de la clase derivada en lugar de hacerlo en la lista de iniciadores?

```
CCuentaAhorro::CCuentaAhorro(string nom, string cue, double sal,
                               double tipo, double mant)
{
    CCuenta(nom, cue, sal, tipo);
    asignarCuotaManten(mant);
}
```

La respuesta es no. Como ya se ha explicado, si el constructor fuera el de este ejemplo, primero invocaría al constructor *CCuenta* sin argumentos y después ejecutaría el código escrito en el cuerpo del mismo; la primera invocaría al constructor *CCuenta* creando un objeto temporal sin ningún efecto y la segunda línea asignaría al objeto *CCuentaAhorro* la cuota de mantenimiento especificada.

COPIA DE OBJETOS

La copia de objetos queda definida mediante el constructor copia y el operador de asignación, implícitos o explícitos (implícitos en el ejemplo que estamos desarrollando) como se puede ver en el ejemplo siguiente:

```
int main()
{
    CCuentaAhorro cuenta01;
    CCuentaAhorro cuenta02("cliente02", "1111111111",
                           200000, 1.75, 10);

    // ...
    cuenta01 = cuenta02; // operador de asignación
    CCuentaAhorro cuenta03 = cuenta02; // constructor copia
    // ...
}
```

¿Cómo son el constructor copia y el operador de asignación de la clase *CCuentaAhorro*? Si hubiéramos definido estos métodos explícitamente, los tendríamos que haber escrito así:

```

CCuentaAhorro::CCuentaAhorro(const CCuentaAhorro& ca) : CCuenta(ca)
{
    cuotaMantenimiento = ca.cuotaMantenimiento;
}

CCuentaAhorro& CCuentaAhorro::operator=(const CCuentaAhorro& ca)
{
    CCuenta::operator=(ca);
    cuotaMantenimiento = ca.cuotaMantenimiento;
    return *this;
}

```

Los métodos expuestos muestran la sintaxis que tendremos que utilizar siempre que en una clase derivada necesitemos implementar el constructor copia y el operador de asignación.

También, según lo estudiado en el capítulo de clases, el constructor copia podría escribirse así:

```

CCuentaAhorro::CCuentaAhorro(const CCuentaAhorro& ca) : CCuenta(ca)
{
    *this = ca; // invoca al operador de asignación
}

```

Analizando estos métodos observamos que el constructor copia de la clase derivada a través de la lista de iniciadores invoca al constructor copia de su clase base, pasándole como argumento una referencia al objeto a copiar, ya que es este constructor el que tiene que llevar a cabo la copia de los atributos heredados de la clase base. Si no hubiéramos especificado esta llamada, se invocaría al constructor sin argumentos de la clase base dando lugar a una copia errónea.

Igualmente, el operador de asignación invoca al operador de asignación de su clase base. Éste es el comportamiento seguido por estos métodos cuando están implícitos y, por lo tanto, el que deben seguir cuando se definen explícitamente.

¿Cómo son el constructor copia y el operador de asignación de la clase *CCuenta*? Si hubiéramos definido estos métodos explícitamente, los tendríamos que haber escrito así:

```

CCuenta::CCuenta(const CCuenta& c)
{
    nombre = c.nombre;
    cuenta = c.cuenta;
    saldo = c.saldo;
    tipoDeInteres = c.tipoDeInteres;
}

```

```

CCuenta& CCuenta::operator=(const CCuenta& c)
{
    nombre = c.nombre;
    cuenta = c.cuenta;
    saldo = c.saldo;
    tipoDeInteres = c.tipoDeInteres;
    return *this;
}

```

Entonces, ¿qué ocurre cuando se ejecuta una línea como la que se muestra sombreada a continuación?

```

CCuentaAhorro cuenta02("cliente02", "1111111111",
                      200000, 1.75, 10);
CCuentaAhorro cuenta03 = cuenta02;

```

1. Se invoca al constructor copia *CCuentaAhorro* pasando como argumento el objeto *cuenta02*. Su parámetro *ca* referencia a este objeto.
2. Se invoca al constructor copia *CCuenta* pasando como argumento el objeto *CCuentaAhorro* referenciado por *ca*. Su parámetro *c* referencia a este objeto. Pero *c* es una referencia a un objeto *CCuenta* y todo ha funcionado correctamente. Esto es debido a que existe una conversión implícita entre objetos, referencias o punteros a objetos de la clase derivada a sus correspondientes de la clase base. En el ejemplo, la referencia *ca* a *CCuentaAhorro* es convertida en una referencia a *CCuenta*.

El ejemplo siguiente utiliza el operador de asignación para copiar el objeto *cuenta02* en *cuenta01*. La explicación de cómo sucede es análoga a la anterior.

```

CCuentaAhorro cuenta01;
CCuentaAhorro cuenta02("cliente02","1111111111", 200000, 1.75, 10);
// ...
cuenta01 = cuenta02; // invoca a CCuentaAhorro::operator=

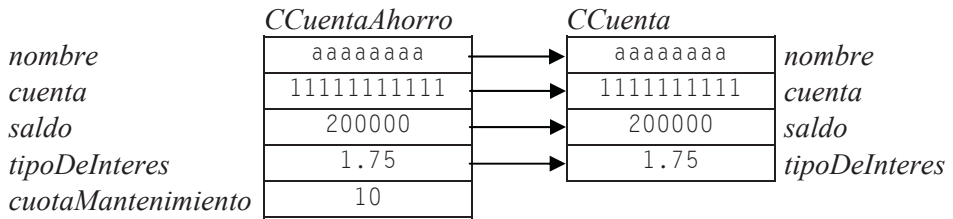
```

También, un objeto de una clase base pública puede ser iniciado con un objeto de una clase derivada de ella invocando a su constructor copia, o bien invocando a su operador de asignación si se trata de una copia sobre un objeto existente. Sin embargo, la asignación inversa no es posible, puesto que la clase derivada tiene miembros que la clase base no tiene. Por ejemplo:

```

CCuenta cuenta01;
CCuentaAhorro cuenta02("aaaaaaa","1111111111", 200000, 1.75, 10);
cuenta01 = cuenta02; // cuenta01.CCuenta::operator=(cuenta02)
CCuenta cuenta03 = cuenta02; // invoca a CCuenta(cuenta02)

```



DESTRUCTORES DE CLASES DERIVADAS

El destructor de una clase base no es heredado por sus clases derivadas. En cuanto a cómo se destruyen los objetos de las clases derivadas diremos que son destruidos en el orden inverso a como son construidos. Esto es, primeramente se ejecuta el cuerpo del destructor de la clase derivada, después son llamados los destructores para sus miembros y por último se ejecuta el destructor de la clase base. Como los destructores no pasan argumentos, no requieren una sintaxis especial.

Por ejemplo, en la aplicación expuesta anteriormente, el destructor de la clase *CCuenta* está definido de la forma siguiente:

```
~CCuenta() { /* no hace nada */ }
```

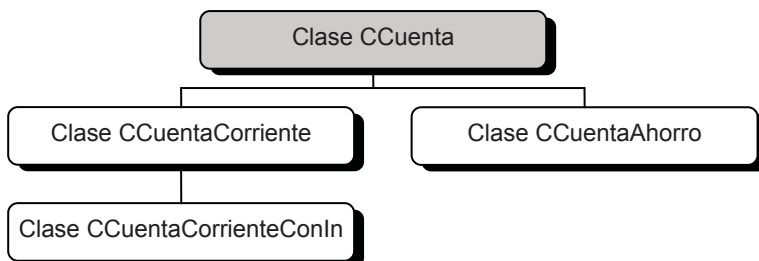
Análogamente, el destructor de la clase *CCuentaAhorro* está definido así:

```
~CCuentaAhorro() { /* no hace nada */ }
```

Siempre que se destruya un objeto de la clase *CCuentaAhorro*, primero se ejecutará el cuerpo de *~CCuentaAhorro*, después los destructores de sus atributos, en este caso de *cuotaMantenimiento* de tipo *int*, y por último se ejecutará el destructor *~CCuenta*.

JERARQUÍA DE CLASES

Una clase derivada puede así mismo ser una clase base de otra clase, y así sucesivamente. En la siguiente figura se puede ver esto con claridad:



El conjunto de clases así definido da lugar a una *jerarquía de clases*. Cuando cada clase derivada lo es de una sola clase base, la estructura jerárquica recibe el nombre de árbol de clases.

La raíz del árbol es la clase que representa el tipo más general y las clases terminales en el árbol (nodos hoja) representan los tipos más especializados.

Las reglas que podemos aplicar para diseñar la clase *CCuentaCorriente* derivada de la clase base *CCuenta* o la clase *CCuentaCorrienteConIn* derivada de la clase base *CCuentaCorriente* son las mismas que hemos aplicado anteriormente para diseñar la clase *CCuentaAhorro* derivada de la clase base *CCuenta*, y lo mismo diremos para cualquier otra clase derivada que deseemos añadir. Esto quiere decir que para implementar una clase derivada como *CCuentaCorrienteConIn*, nos es suficiente con conocer a fondo su clase base *CCuentaCorriente* sin importarnos *CCuenta*.

Observe que la clase *CCuenta* actúa como clase base de más de una clase, concretamente de las clases *CCuentaAhorro* y *CCuentaCorriente*.

Como ejemplo, vamos a completar la jerarquía de clases expuesta con las clases que faltan: *CCuentaCorriente* y *CCuentaCorrienteConIn*.

La clase *CCuentaCorriente* es una nueva clase que hereda de la clase *CCuenta*. Esto implica que la funcionalidad de la misma estará soportada por todos los miembros heredados de su clase base más los que añadamos, que van a ser los siguientes:

Atributo	Significado
<i>transacciones</i>	Dato de tipo int que almacena el número de transacciones efectuadas sobre esa cuenta.
<i>importePorTrans</i>	Dato de tipo double que almacena el importe que la entidad bancaria cobrará por cada transacción.
<i>transExentas</i>	Dato de tipo int que almacena el número de transacciones gratuitas.
Método	Significado
<i>CCuentaCorriente</i>	Es el constructor de la clase. Inicia los atributos de la misma.
<i>decrementarTransacciones</i>	Decrementa en 1 el número de transacciones.
<i>asignarImportePorTrans</i>	Establece el importe por transacción. Retorna false si el importe es negativo, y true en otro caso.
<i>obtenerImportePorTrans</i>	Devuelve el importe por transacción.

Método	Significado
<i>asignarTransExentas</i>	Establece el número de transacciones exentas. Retorna false si el número de transacciones es negativo, y true en otro caso.
<i>obtenerTransExentas</i>	Devuelve el número de transacciones exentas.
<i>ingreso</i>	Añade la cantidad especificada al saldo actual de la cuenta e incrementa el número de transacciones.
<i>reintegro</i>	Resta la cantidad especificada del saldo actual de la cuenta e incrementa el número de transacciones.
<i>comisiones</i>	Se ejecuta el día 1 de cada mes para cobrar el importe de las transacciones efectuadas que no estén exentas y pone el número de transacciones a cero.
<i>intereses</i>	Se ejecuta el día 1 de cada mes para calcular el importe correspondiente a los intereses/mes producidos y añadirlo al saldo. Hasta 3.000 euros al 0,5%. El resto al interés establecido.

Aplicando la teoría expuesta hasta ahora y procediendo de forma similar a como lo hicimos para construir la clase derivada *CCuentaAhorro*, la definición de la clase *CCuentaCorriente* es la siguiente:

```
// cuenta_corriente.h - Declaración de la clase CCuentaCorriente
#ifndef _CUENTA_CORRIENTE_H_
#define _CUENTA_CORRIENTE_H_
#include "cuenta.h"

class CCuentaCorriente : public CCuenta
{
    // Atributos
private:
    int transacciones;
    double importePorTrans;
    int transExentas;
    // Métodos
public:
    CCuentaCorriente(std::string nom = "sin nombre",
                     std::string cue = "0000",
                     double sal = 0.0, double tipo = 0.0,
                     double imptrans = 0.0, int transex = 0);
    void decrementarTransacciones();
    bool asignarImportePorTrans(double);
    double obtenerImportePorTrans() const;
    bool asignarTransExentas(int);
    int obtenerTransExentas() const;
    void ingreso(double);
    void reintegro(double);
    void comisiones();
};
```

```
        double intereses();
    };
#endif // _CUENTA_CORRIENTE_H_

// cuenta_corriente.cpp - Definición de la clase CCuentaCorriente
#include <iostream>
#include "cuenta_corriente.h"
#include "fecha.h"
using namespace std;

/////////////////////////////////////////////////////////////////
// Clase CCuentaCorriente: clase derivada de CCuenta
//
CCuentaCorriente::CCuentaCorriente(string nom, string cue,
    double sal, double tipo, double imptrans, int transex) :
    CCuenta(nom, cue, sal, tipo),
    importePorTrans(imptrans), transExentas(transex)
{
    // Verificar datos
    transacciones = 0;
    asignarImportePorTrans(imptrans);
    asignarTransExentas(transex);
}

void CCuentaCorriente::decrementarTransacciones()
{
    transacciones--;
}

bool CCuentaCorriente::asignarImportePorTrans(double imptrans)
{
    if (imptrans >= 0)
        importePorTrans = imptrans;
    else
        cout << "Error: cantidad negativa\n";
    return imptrans >= 0;
}

double CCuentaCorriente::obtenerImportePorTrans() const
{
    return importePorTrans;
}

bool CCuentaCorriente::asignarTransExentas(int transex)
{
    if (transex >= 0)
        transExentas = transex;
    else
        cout << "Error: cantidad negativa\n";
    return transex >= 0;
}
```

```
int CCuentaCorriente::obtenerTransExentas() const
{
    return transExentas;
}

void CCuentaCorriente::ingreso(double cantidad)
{
    CCuenta::ingreso(cantidad);
    transacciones++;
}

void CCuentaCorriente::reintegro(double cantidad)
{
    CCuenta::reintegro(cantidad);
    transacciones++;
}

void CCuentaCorriente::comisiones()
{
    // Se aplican mensualmente por el mantenimiento de la cuenta
    int dia, mes, anyo;

    CFecha::obtenerFechaActual(dia, mes, anyo);
    if (dia == 1)
    {
        int n = transacciones - transExentas;
        if (n > 0) reintegro(n * importePorTrans);
        transacciones = 0;
    }
}

double CCuentaCorriente::intereses()
{
    int dia, mes, anyo;
    CFecha::obtenerFechaActual(dia, mes, anyo);
    if (dia != 1) return 0.0;

    // Acumular los intereses por mes sólo los días 1 de cada mes
    double interesesProducidos = 0.0;
    // Hasta 3000 euros al 0.5%. El resto al interés establecido.
    if (estado() <= 3000)
        interesesProducidos = estado() * 0.5 / 1200.0;
    else
    {
        interesesProducidos = 3000 * 0.5 / 1200.0 +
            (estado() - 3000) * obtenerTipoDeInteres() / 1200.0;
    }
    ingreso(interesesProducidos);

    // Este ingreso no debe incrementar las transacciones
    decrementarTransacciones();
}
```



```

    return interesesProducidos;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

Se puede observar que el constructor de la clase *CCuentaCorriente* tiene los parámetros necesarios para iniciar sus datos miembro, excepto *transacciones* que inicialmente vale 0, y los heredados de su clase base. Dicho constructor llama primero al constructor de su clase base y después a los métodos de la propia clase que permiten iniciar de forma segura los atributos de la misma. También se han especificado valores por omisión.

Procediendo de forma similar a como lo hemos hecho para las clases *CCuentaAhorro* y *CCuentaCorriente*, construimos a continuación la clase *CCuentaCorrienteConIn* (cuenta corriente con intereses) derivada de *CCuentaCorriente*.

Supongamos que este tipo de cuenta se ha pensado para incentivar con una rentabilidad mayor respecto a *CCuentaCorriente* a los clientes que conserven un saldo mínimo en su cuenta.

Digamos que se trata de una cuenta de tipo *CCuentaCorriente* que precisa un saldo mínimo de 3.000 euros para que pueda acumular intereses. Según esto, *CCuentaCorrienteConIn*, además de los miembros heredados, sólo precisa implementar sus constructores y variar el método *intereses*:

Método	Significado
<i>CCuentaCorrienteConIn</i>	Es el constructor de la clase. Inicia los atributos de un objeto <i>CCuentaCorrienteConIn</i> .
<i>intereses</i>	Permite calcular el importe/mes correspondiente a los intereses producidos. Precisa un saldo mínimo de 3.000 euros.

La definición correspondiente a esta clase se expone a continuación:

```

// cuenta_corriente+.h - Clase CCuentaCorrienteConIn
#ifdef !_CUENTA_CORRIENTECOIN_H_
#define _CUENTA_CORRIENTECOIN_H_
#include "cuenta_corriente.h"

class CCuentaCorrienteConIn : public CCuentaCorriente
{
    // Métodos
public:
    CCuentaCorrienteConIn(std::string nom = "sin nombre",
        std::string cue = "0000", double sal = 0.0, double tipo = 0.0,
        double imptrans = 0.0, int transex = 0);

```

```

        double intereses();
    };

#endif // _CUENTA_CORRIENTECOIN_H_

// cuenta_corriente+.cpp - Clase CCuentaCorrienteConIn
#include <iostream>
#include "cuenta_corriente+.h"
#include "fecha.h"
using namespace std;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Clase CCuentaCorrienteConIn: clase derivada de CCuentaCorriente
//
// Métodos
CCuentaCorrienteConIn::CCuentaCorrienteConIn(string nom,
        string cue, double sal, double tipo,
        double imptrans, int transex) :
CCuentaCorriente(nom, cue, sal, tipo, imptrans, transex)
{
}

double CCuentaCorrienteConIn::intereses()
{
    int dia, mes, anyo;
    CFecha::obtenerFechaActual(dia, mes, anyo);

    if (dia != 1 || estado() < 3000) return 0.0;

    // Acumular interés mensual sólo los días 1 de cada mes
    double interesesProducidos = 0.0;
    interesesProducidos = estado() * obtenerTipoDeInteres()/1200.0;
    ingreso(interesesProducidos);
    // Este ingreso no debe incrementar las transacciones
    decrementarTransacciones();

    // Devolver el interés mensual por si fuera necesario
    return interesesProducidos;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

La clase *CCuenta* es la “clase base directa” (o simplemente clase base) de *CCuentaAhorro* y de *CCuentaCorriente* y es una “clase base indirecta” para *CCuentaCorrienteConIn*. Cuando la herencia es simple una clase derivada sólo tiene una clase base directa, pero puede tener varias clases base indirectas: todas las que haya en el camino para llegar desde su clase base hasta la clase raíz; esto es importante porque lo que una clase derivada hereda de su clase base será heredado a su vez por una clase derivada de ella, y así sucesivamente.

Una clase derivada que redefine un método heredado tiene acceso a su propia versión y a las publicadas por sus clases base directas e indirectas. Por ejemplo, las clases *CCuenta* y *CCuentaCorriente* incluyen cada una su versión del método *ingreso* y la clase derivada *CCuentaCorrienteConIn* hereda el método *ingreso* de *CCuentaCorriente*. Entonces, *CCuentaCorrienteConIn*, además de a su propia versión, puede acceder a la versión de su clase base directa por medio de la expresión *CCuentaCorriente::ingreso* (en este caso ambas versiones son la misma) y también puede acceder a la versión de su clase base indirecta *CCuenta* por medio de la expresión *CCuenta::ingreso*.

Según lo expuesto, las líneas de código:

```
ingreso(interesesProducidos);
decrementarTransacciones();
```

del método *intereses* de la clase *CCuentaCorriente* podrían ser sustituidas por la indicada a continuación, puesto que el método *ingreso* de *CCuenta* no actúa sobre las transacciones:

```
CCuenta::ingreso(interesesProducidos);
```

Ídem para el método *intereses* de la clase *CCuentaCorrienteConIn*.

A continuación se presenta una aplicación con algunos ejemplos de operaciones con objetos de las clases pertenecientes a la jerarquía construida:

```
// test.cpp - Operaciones con la jerarquía de clases de CCuenta
#include <iostream>
#include "cuenta_ahorro.h"
#include "cuenta_corriente+.h"
using namespace std;

void visualizar(CCuentaAhorro&);

int main()
{
    CCuentaAhorro cuenta01(
        "Un nombre", "Una cuenta", 10000, 3.5, 10);
    visualizar(cuenta01);

    CCuentaCorrienteConIn cuenta02;
    cuenta02.asignarNombre("cliente 02");
    cuenta02.asignarCuenta("1234567890");
    cuenta02.asignarTipoDeInteres(3.0);
    cuenta02.asignarTransExentas(0);
    cuenta02.asignarImportePorTrans(0.01);
```

```
    cuenta02.ingreso(20000);
    cuenta02.reintegro(5000);
    cuenta02.intereses();
    cuenta02.comisiones();
    cout << cuenta02.obtenerNombre() << endl;
    cout << cuenta02.obtenerCuenta() << endl;
    cout << cuenta02.estado() << endl;
}

void visualizar(CCuentaAhorro& cuenta)
{
    cout << cuenta.obtenerNombre() << endl;
    cout << cuenta.obtenerCuenta() << endl;
    cout << cuenta.estado() << endl;
    cout << cuenta.obtenerTipoDeInteres() << endl;
    cout << cuenta.intereses() << endl;
}
```

En la aplicación anterior se puede observar cómo el método **main** construye dos objetos: *cuenta01* de la clase *CCuentaAhorro* y *cuenta02* de la clase *CCuentaCorrienteConIn*. Para construir *cuenta01* se ha utilizado el constructor *CCuentaAhorro* con argumentos. Una vez construido, obsérvese que responde a una serie de mensajes ejecutando los métodos del mismo nombre, unos heredados de su clase base, como *obtenerNombre*, y otros propios, como *intereses*. En cambio, para construir *cuenta02* se ha utilizado el constructor *CCuentaCorrienteConIn* sin argumentos. Una vez construido, se puede también observar que responde a una serie de mensajes ejecutando los métodos del mismo nombre, unos heredados de su clase base directa, como *reintegro*, otros heredados de su clase base indirecta, como *asignarNombre*, y otros propios, como *intereses*.

Finalmente, indicar que aunque en ninguna clase de nuestra jerarquía han intervenido miembros **static**, su comportamiento en cuanto a la herencia se refiere es el mismo que el de los otros miembros, pero teniendo presente que son miembros de la clase; y si es necesario, cuando se trate de métodos, también pueden ser redefinidos, aunque, en este caso, el nombre de la clase indicará la versión del método que se invocará. Una advertencia: si definiera, por ejemplo, en *CCuenta* el atributo *tipoDeInteres* **static**, lógicamente se mantendría una única copia que utilizarían tanto los objetos de *CCuenta* como los de sus clases derivadas.

FUNCIONES AMIGAS

Supongamos una clase base y su derivada. Cada una de ellas, además de sus miembros públicos, protegidos y privados, aporta una función amiga:

```
class Base
{
```

```

friend void FnAmigaDeBase();
private:
    void mPrivadoDeBase(){}
protected:
    void mProtegidoDeBase(){}
public:
    void mPublicoDeBase(){}
};

class Derivada : public Base
{
    friend void FnAmigaDeDerivada();
private:
    void mPrivadoDeDerivada(){}
protected:
    void mProtegidoDeDerivada(){}
public:
    void mPublicoDeDerivada(){}
};

```

Teniendo presente que un método no **static** de una clase siempre tiene que ser invocado para un objeto de la misma o de alguna de sus derivadas y recordando la definición de función **friend**, ¿a qué miembros de la clase base y de la derivada puede acceder la función *FnAmigaDeBase*?

Es fácil adivinar que una función **friend** de la clase base podrá acceder a los miembros de dicha clase, los cuales serán heredados por la clase derivada y, además, como cualquier otro método de la misma, a los miembros públicos de la clase derivada, pero no a sus miembros protegidos y privados, lo cual indica que la amistad no se hereda. Esto puede probarse fácilmente ejecutando el código siguiente:

```

void FnAmigaDeBase()
{
    Derivada objd;
    objd.mPrivadoDeBase();           // correcto
    objd.mProtegidoDeBase();        // correcto
    objd.mPublicoDeBase();          // correcto
    objd.mPrivadoDeDerivada();      // error: no se puede acceder
    // a un miembro privado de la clase Derivada
    objd.mProtegidoDeDerivada();    // error: no se puede acceder
    // a un miembro protegido de la clase Derivada
    objd.mPublicoDeDerivada();      // correcto
}

```

Y, ¿a qué miembros de la clase base y de la derivada puede acceder la función *FnAmigaDeDerivada*?

Si la función es amiga (**friend**) de la clase derivada, entonces podrá acceder a los miembros de dicha clase y, además, como cualquier otro método de la misma, a los miembros protegidos y públicos de la clase base, pero no a sus miembros privados. También esto puede probarse fácilmente ejecutando el código siguiente:

```
void FnAmigaDeDerivada()
{
    Derivada objd;
    objd.mPrivadoDeBase();      // error: no se puede acceder
    // a un miembro privado de la clase Base
    objd.mProtegidoDeBase();    // correcto
    objd.mPublicoDeBase();     // correcto
    objd.mPrivadoDeDerivada(); // correcto
    objd.mProtegidoDeDerivada(); // correcto
    objd.mPublicoDeDerivada(); // correcto
}
```

PUNTEROS Y REFERENCIAS

Los punteros y las referencias a objetos de una clase derivada pueden ser declarados y manipulados de la misma forma que los punteros y referencias a objetos de una clase cualquiera, tal y como ya expusimos en capítulos anteriores. Veamos algunos ejemplos basados en la jerarquía de clases que acabamos de construir:

```
int main()
{
    CCuentaCorriente cuenta01("cliente01", "1234567890",
                               10000, 3.5, 30);
    fa(&cuenta01); //--> CCuentaCorriente *p = &cuenta01; fa(p);
    fb(cuenta01); //--> CCuentaCorriente& r = cuenta01; fb(r);
}

void fa(CCuentaCorriente *p)
{
    string cuenta = p->obtenerCuenta();
    double saldo = p->estado();
    // ...
}

void fb(CCuentaCorriente& r)
{
    string cuenta = r.obtenerCuenta();
    double saldo = r.estado();
    // ...
}
```

La función **main** de este ejemplo declara un objeto *cuenta01* de la clase *CCuentaCorriente* derivada de *CCuenta*. Después invoca a la función *fa* que define un puntero *p* al objeto *cuenta01* pasado como argumento y finalmente invoca a la función *fb* que define una referencia *r* al objeto *cuenta01* pasado como argumento. Una vez que disponemos del puntero o de la referencia a un objeto podemos trabajar con él como lo hemos venido haciendo hasta ahora, según muestran las funciones *fa* y *fb*.

Conversiones implícitas

El ejemplo anterior no aporta nada que nos sorprenda; operaciones como éstas ya han sido expuestas anteriormente. Pero, ¿qué pasaría si *cuenta01* fuera un objeto de la clase *CCuentaCorrienteConIn* derivada de *CCuentaCorriente*? Por ejemplo:

```
int main()
{
    CCuentaCorrienteConIn cuenta01("cliente01","1234567890",
                                    10000, 3.5, 1.0, 6);
    fa(&cuenta01);
    fb(cuenta01);
}
```

Si ejecutamos este ejemplo, comprobaremos que los resultados obtenidos son los mismos que obtuvimos con el ejemplo anterior. Esto es así porque C++ permite convertir implícitamente un puntero o una referencia a un objeto de una clase derivada, en un puntero o una referencia a su clase base directa o indirecta. Veamos otro ejemplo:

```
int main()
{
    CCuentaCorriente cuenta01("cliente01", "1234567891",
                               10000, 3.5, 1.0, 6);
    CCuentaCorrienteConIn cuenta02("cliente02", "1234567892",
                                    20000, 2.0, 1.0, 6);
    // Conversiones de derivada a base
    fa(&cuenta01); fa(&cuenta02);
    fb(cuenta01); fb(cuenta02);
}

void fa(CCuenta *p)
{
    string cuenta = p->obtenerCuenta();
    string nombre = p->obtenerNombre();
    // ...
}
```

```
void fb(CCuenta& r)
{
    string cuenta = r.obtenerCuenta();
    string nombre = r.obtenerNombre();
    // ...
}
```

En el ejemplo anterior las funciones *fa* y *fb* declaran, respectivamente, un puntero y una referencia a un objeto *CCuenta*, los cuales utilizamos después para referenciar indistintamente a un objeto *cuenta01* de la clase *CCuentaCorriente* o a un objeto *cuenta02* de la clase *CCuentaCorrienteConIn*.

Cuando accedemos a un objeto por medio de una variable no del tipo del objeto, sino del tipo de alguna de sus clases base (directas o indirectas) según muestra el ejemplo anterior, es el tipo de la variable el que determina qué mensajes puede recibir el objeto referenciado y, por lo tanto, qué métodos pueden ser invocados por éste. ¿Cuáles son esos métodos? Pues los correspondientes al tipo de la variable que utilizamos para hacer referencia al objeto, no los de la clase del objeto.

Resumiendo: cuando accedemos a un objeto de una clase derivada por medio de un puntero o una referencia a su clase base, ese objeto sólo puede ser manipulado por los métodos de su clase base. Por ejemplo, modifiquemos las funciones *fa* y *fb* como se muestra a continuación (**main** no se modifica):

```
void fa(CCuenta *p)
{
    p->asignarImportePorTrans(1.0);
    p->asignarTransExentas(10);
    // ...
}

void fb(CCuenta& r)
{
    r.asignarImportePorTrans(1.0);
    r.asignarTransExentas(10);
    // ...
}
```

Este último ejemplo sigue la misma pauta que el anterior. Pero ahora observamos en ambas funciones que un intento de acceder al método *asignarImportePorTrans* ocasiona un error. Esto es porque el tipo de la variable, puntero o referencia a *CCuenta*, determina que el objeto referenciado sólo puede recibir mensajes de la clase de dicha variable; dicho de otra forma, sólo puede ser manipulado por métodos de la clase *CCuenta* (propios y heredados). Lo mismo diríamos respecto al mensaje *asignarTransExentas*.

Así mismo, cuando se invoca a un método que está definido en la clase base y redefinido en sus clases derivadas, la versión que se ejecuta depende también de la clase del puntero o de la referencia, no del tipo del objeto referenciado. Por ejemplo, modifiquemos otra vez las funciones *fa* y *fb* como se muestra a continuación (**main** no se modifica):

```
void fa(Cuenta *p)
{
    double intereses = p->intereses();
    // ...
}

void fb(Cuenta& r)
{
    double intereses = r.intereses();
    // ...
}
```

En el ejemplo anterior las funciones *fa* y *fb* declaran, respectivamente, un puntero y una referencia a un objeto *Cuenta*, que utilizamos después para referenciar indistintamente a un objeto *cuenta01* de la clase *CuentaCorriente* o a un objeto *cuenta02* de la clase *CuentaCorrienteConIn*. Por otra parte, el método *intereses* está definido en la clase base *Cuenta* y redefinido en sus clases derivadas *CuentaCorriente* y *CuentaCorrienteConIn*, pero observamos que independientemente del objeto referenciado, las expresiones *p->intereses()* o *r.intereses()* invocan a *Cuenta::intereses()*.

Restricciones

Anteriormente dijimos que los especificadores de acceso para una clase base también controlaban la conversión de punteros y de referencias desde el tipo de la clase derivada al de la clase base. Consideremos una clase *CD* derivada de una clase base *CB*:

- Si *CB* es una clase base *privada*, sólo los métodos y funciones amigas de *CD* pueden convertir un *CD ** a *CB **.
- Si *CB* es una clase base *protegida*, sólo los métodos y funciones amigas de *CD* y los métodos y funciones amigas de las clases derivadas de *CD* pueden convertir un *CD ** a *CB **.
- En cambio, no hay restricciones si *CB* es una clase base *pública*; en este caso, cualquier función puede convertir un *CD ** a *CB **.

Saltarse estas restricciones da lugar a errores que serán detectados por el compilador.

Conversiones explícitas

La conversión contraria, esto es, de un puntero o una referencia a un objeto de la clase base a un puntero o a una referencia a su clase derivada, se puede hacer, pero forzando dicha conversión mediante el operador **static_cast**. Este tipo de conversiones, en ocasiones, puede conducir a situaciones absurdas, por lo que se recomienda no hacerlas si no es utilizando el operador **dynamic_cast** (conversiones con verificación durante la ejecución). Por ejemplo:

```
int main()
{
    CCuenta cuenta01("cliente01", "1234567891", 10000, 3.5);
    // Conversiones de base a derivada
    fa(static_cast<CCuentaCorriente *>(&cuenta01));
    fb(static_cast<CCuentaCorriente&>(cuenta01));
}

void fa(CCuentaCorriente *p)
{
    string nombre = p->obtenerNombre();
    // ...
}

void fb(CCuentaCorriente& r)
{
    string nombre = r.obtenerNombre();
    // ...
}
```

Conversiones como las realizadas en el ejemplo anterior pueden resultar peligrosas, porque no se puede asegurar qué tipo de objeto está referenciado por el puntero o por la referencia. Para aclarar este punto, obsérvese el ejemplo anterior: *cuenta01* es un objeto de la clase *CCuenta* que pasará a estar referenciado, en un caso, por el parámetro *p* de *fa* que es un puntero a *CCuentaCorriente* y, en otro caso, por el parámetro *r* de *fb* que es una referencia a *CCuentaCorriente*. Al ejecutar este ejemplo, se observa que todo funciona correctamente, ya que el dato obtenido *nombre* es un atributo de *cuenta01*. Pero supongamos que *fa* y *fb* estuvieran definidas de esta otra forma:

```
void fa(CCuentaCorriente *p)
{
    string nombre = p->obtenerNombre();
    int te = p->obtenerTransExentas();
```

```

    // ...
}

void fb(CCuentaCorriente& r)
{
    string nombre = r.obtenerNombre();
    int te = r.obtenerTransExentas();
    // ...
}

```

Igual que antes, tanto *p* como *r* hacen referencia al objeto *cuenta01* de *CCuenta*. Pero ahora, cuando se invoque al método *obtenerTransExentas*, éste intentará acceder al dato miembro *transExentas* que ese objeto no tiene, y el resultado será impredecible.

Este tipo de situaciones puede ser detectado con el operador **dynamic_cast**. Por ejemplo:

```

fa(dynamic_cast<CCuentaCorriente *>(&cuenta01));
fb(dynamic_cast<CCuentaCorriente&>(cuenta01));

```

En este caso, el compilador, o bien avisa de que esas conversiones no se pueden realizar y detiene la compilación, o bien, si permite realizar la compilación, en el primer caso (conversión de un puntero de base a derivada) **dynamic_cast** devolverá durante la ejecución un puntero nulo, y en el segundo (conversión de una referencia de base a derivada) lanzará una excepción **bad_cast**. Según esto, la función **main** anterior podríamos escribirla de forma segura así:

```

int main()
{
    CCuenta cuenta01("cliente01", "1234567891", 10000, 3.5);
    // Conversiones de base a derivada
    CCuentaCorriente * p;

    if (p = dynamic_cast<CCuentaCorriente *>(&cuenta01))
        fa(p);
    else
        cout << "conversión entre punteros no válida\n";

    try
    {
        fb(dynamic_cast<CCuentaCorriente&>(cuenta01));
    }
    catch(bad_cast)
    {
        cout << "conversión entre referencias no válida\n";
    }
}

```

Para poder realizar una conversión descendente (de base a derivada) el operador **dynamic_cast** necesita un puntero o una referencia a un tipo polimórfico (clase con métodos virtuales), concepto que estudiaremos un poco más adelante en este mismo capítulo.

MÉTODOS VIRTUALES

En los apartados anteriores hemos visto que cuando se invoca a un método que está definido en la clase base y redefinido en sus clases derivadas, la versión que se ejecuta depende del tipo del objeto, del tipo del puntero o del tipo de la referencia que se utilice para invocar al mismo.

Por ejemplo, si echamos una ojeada a la clase *CCuenta* y a sus clases derivadas, definidas anteriormente, observamos que el método *intereses* de la clase *CCuenta* ha sido redefinido en todas sus clases derivadas, directas o indirectas (el método *comisiones* también, excepto en *CCuentaCorrienteConIn*). Si ahora ejecutamos el código siguiente:

```
int main()
{
    CCuentaCorriente cuenta01("cliente01", "1234567891",
                              10000, 3.5, 1.0, 6);
    fa(&cuenta01);
    fb(cuenta01);
}

void fa(CCuenta *p)
{
    double intereses = p->intereses();
    // ...
}

void fb(CCuenta& r)
{
    double intereses = r.intereses();
    // ...
}
```

el comportamiento del compilador es el esperado, aunque quizás no el deseado, ya que en ambos casos el método *intereses* que se ejecuta pertenece a la clase *CCuenta*, que como podemos observar es el tipo del puntero *p* y de la referencia *r*, cuando quizás deseábamos que se ejecutase el método *intereses* de la clase del objeto referenciado.

La solución al problema planteado pasa porque sea el mismo sistema el que se encargue de la identificación durante la ejecución de la clase de los objetos apuntados, mecanismo que C++ proporciona por medio de los métodos *virtuales*.

Un método *virtual* es un miembro de una clase base que puede ser redefinido en cada una de las clases derivadas de ésta, y una vez redefinido puede ser accedido mediante un puntero o una referencia a la clase base, resolviéndose la llamada en función del tipo del objeto referenciado. Una clase con métodos virtuales se denomina *tipo polimórfico*.

Un método se declara *virtual* escribiendo la palabra clave **virtual** al principio de la declaración del método en la clase donde aparece por primera vez. Las redefiniciones que realicemos de este método en las clases derivadas no necesitan incorporar en su declaración la palabra clave **virtual**, porque ya son declarados implícitamente métodos virtuales; hacerlo sería redundante.

La redefinición de un método virtual en una clase derivada debe tener el mismo nombre, número y tipos de parámetros y tipo del valor retornado que en la clase base; en otro caso, se producirá un error (hay una excepción que comentaremos más adelante al hablar de constructores virtuales). Además, una clase derivada puede contener sus propios métodos virtuales; esto es, métodos virtuales no heredados de sus clases base.

Según lo expuesto, para declarar virtual los métodos *comisiones* e *intereses* de la clase *CCuenta*, clase raíz de la jerarquía de clases que hemos construido en este capítulo, edite el fichero *cuenta.h* y proceda como se indica a continuación:

```
class CCuenta
{
    // Atributos
    private:
        // ...
    // Métodos
    public:
        // ...
        virtual void comisiones();
        virtual double intereses();
        // ...
};
```

Obsérvese que tanto el método *comisiones* como *intereses* se han declarado virtuales en la clase donde se declaran por primera vez.

Una vez realizada la modificación anterior, volvemos a ejecutar el ejemplo anterior y comprobaremos que:

p->intereses() llama al método *CCuentaCorriente::intereses*
r.intereses() llama al método *CCuentaCorriente::intereses*

Por lo tanto, el método invocado pertenece, como deseábamos, a la misma clase que el objeto referenciado por *p* o por *r* (en el ejemplo, el objeto referenciado es *cuenta01* de la clase *CCuentaCorriente*). Quiere esto decir que el mecanismo *virtual* garantiza que el objeto será manipulado por los métodos de su clase.

Si una clase derivada no provee una redefinición de un método declarado virtual en su clase base, una llamada al mismo hace que se ejecute el definido en su clase base. Por ejemplo, si *CCuentaCorriente* no redefiniera el método *intereses*, la llamada *p->intereses()* del ejemplo anterior invocaría al método *intereses* de la clase *CCuenta*. Éste es el motivo de por qué un método virtual se declara en la clase base. Dicho de otra forma, cuando la clase derivada no redefine el método virtual de su clase base, hereda la implementación de la clase base, de manera que una llamada al mismo a través de un objeto de la clase derivada hace que se ejecute el método virtual de su clase base.

Resumiendo:

- Una llamada a un método virtual se resuelve siempre en función del tipo del objeto referenciado.
- Una llamada a un método normal (no virtual) se resuelve en función del tipo del puntero o de la referencia.
- Una llamada a un método virtual específico exige utilizar el operador `::` de resolución del ámbito. Esta forma de llamar a un método virtual suprime el mecanismo *virtual*.

El siguiente ejemplo muestra de una forma práctica cómo se ejecutan las llamadas a métodos virtuales y no virtuales a través de punteros.

```
// virtual.cpp - Métodos virtuales y no virtuales
#include <iostream>
using namespace std;

class CB
{
public:
    virtual void mVirtual1(); // método virtual
    void mNoVirtual();      // método no virtual
};

void CB::mVirtual1()
{
```

```
    cout << "método virtual 1 en CB\n";
}
void CB::mNoVirtual()
{
    cout << "método no virtual en CB\n";
}
```

```
class CD1 : public CB
{
public:
    void mVirtual1();           // método virtual
    virtual void mVirtual2(); // método virtual
    void mNoVirtual();         // método no virtual
};
```

```
void CD1::mVirtual1()
{
    cout << "método virtual 1 en CD1\n";
}
void CD1::mVirtual2()
{
    cout << "método virtual 2 en CD1\n";
}
void CD1::mNoVirtual()
{
    cout << "método no virtual en CD1\n";
}
```

```
class CD2 : public CD1
{
public:
    void mVirtual1(); // método virtual
    void mVirtual2(); // método virtual
    void mNoVirtual(); // método no virtual
};
```

```
void CD2::mVirtual1()
{
    cout << "método virtual 1 en CD2\n";
}
void CD2::mVirtual2()
{
    cout << "método virtual 2 en CD2\n";
}
void CD2::mNoVirtual()
{
    cout << "método no virtual en CD2\n";
}
```

```
void fx(CB *p) // función externa
{
```

```

    p->mVirtual1();
    // ...
}

```

```

int main()
{
    CB *p1CB = new CD1; // puntero a CB que apunta a un objeto CD1
    CD1 *p1CD1 = new CD2; // puntero a CD1 que apunta a un objeto CD2
    CB *p2CB = new CD2; // puntero a CB que apunta a un objeto CD2

    // Llamadas a los métodos
    p1CB->mVirtual1(); // llama a CD1::mVirtual1
    p1CB->mNoVirtual(); // llama a CB::mNoVirtual
    p1CB->CB::mVirtual1(); // llama a CB::mVirtual1
    p1CD1->mVirtual2(); // llama a CD2::mVirtual2
    p1CD1->mNoVirtual(); // llama a CD1::mNoVirtual
    p1CD1->CD1::mVirtual2(); // llama a CD1::mVirtual2
    fx(p2CB); // llama a CD2::mVirtual1

    delete p1CB;
    delete p1CD1;
    delete p2CB;
}

```

Observe cómo una llamada a un método virtual se resuelve en función del tipo del objeto apuntado, cómo una llamada a un método no virtual se resuelve en función del tipo del puntero o de la referencia y cómo una llamada explícita, utilizando el operador `::` de resolución del ámbito, a un método virtual suprime el mecanismo virtual. Fíjese también que la clase *CDI* define su propio método virtual, además de redefinir el método virtual heredado.

Una función externa o **static** no puede ser declarada **virtual**, ya que un método virtual sólo es llamado para objetos de su clase. Sin embargo, un método virtual sí puede ser declarado **friend** de otra clase.

Cómo son implementados los métodos virtuales

Cuando se compile el ejemplo *virtual.cpp* anterior, el compilador no podrá identificar el método que va a ser llamado por una sentencia como `p->mVirtual1()`, ya que puede ser cualquiera de varios métodos diferentes. Esto se debe, como sabemos, a que el método `mVirtual1` fue declarado **virtual** en la clase base *CB*, lo cual supone que pueda haber múltiples formas de él, una por cada clase derivada directa o indirectamente de *CB*.

Por lo tanto, el compilador debe añadir código que permita evaluar la sentencia durante la ejecución, que es cuando se puede conocer a qué tipo de objeto

apunta p , lo que permitirá saber a qué método hay que invocar. Esto es conocido como *ligadura dinámica* o *ligadura retrasada*. Esta forma de actuar es muy diferente a la de las funciones externas de C++ o a la de los métodos no virtuales de una clase. En ambos casos, la sentencia de llamada al método es convertida durante la compilación en un salto a una dirección fija, coincidente con el punto de entrada al método. Esto es conocido como *ligadura estática* o *ligadura al principio*.

En términos de la POO diremos que un mensaje dirigido a un objeto se asocia con un método. Cuando la asociación se hace durante la compilación, se denomina ligadura estática y cuando se hace durante la ejecución, se denomina ligadura dinámica. Ésta última tiene lugar cuando el método que se asocia es virtual, lo cual permite retrasar hasta el momento de la ejecución la decisión de qué método concreto debe ejecutarse.

En algunas situaciones, una llamada a un método virtual puede ser compilada como una llamada a un método no virtual; esto es, utilizando una ligadura estática. Por ejemplo:

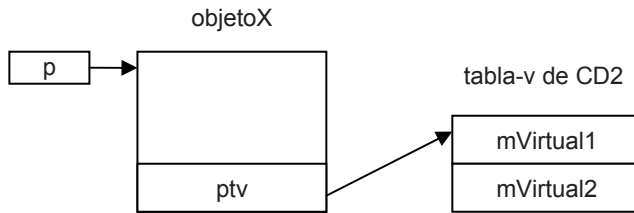
```
CD2 objetoX;
CB *p = &objetoX;

objetoX.mVirtual1(); // ligadura estática
p->mVirtual1();      // posible ligadura estática
```

En este ejemplo, el tipo de *objetoX* es conocido y el tipo del objeto apuntado por p también. Por lo tanto, queda perfectamente determinado qué método *mVirtual1* hay que llamar. Quiere esto decir que el compilador utilizará una ligadura dinámica cuando no pueda utilizar una ligadura estática, como ocurre en la función fx del ejemplo anterior.

La ligadura dinámica en C++ se implementa a través de una tabla de métodos virtuales o *tabla-v*. Dicha tabla está formada por una matriz de punteros a métodos, que el compilador asocia a cada una de las clases que contienen uno o más métodos virtuales. Por ejemplo, en la jerarquía de clases construida anteriormente en el ejemplo *virtual.cpp*, *CB*, *CD1* y *CD2* tienen cada una de ellas su propia *tabla-v*. Esta tabla contiene un puntero a un método por cada uno de los métodos virtuales de la clase. Uno de estos métodos virtuales puede ser un método definido en dicha clase o un método heredado directa o indirectamente de una clase base. Así, la *tabla-v* de *CD2* tiene un puntero al método *mVirtual1* y otro a *mVirtual2*.

Según lo expuesto, cada objeto de una clase contiene un puntero oculto, *ptv*, a la tabla de métodos virtuales de dicha clase.



De esta forma, cuando se compila una llamada a un método virtual como:

```
p->mVirtual1();
```

el compilador transforma dicha llamada en una llamada indirecta al método, la cual, utilizando el puntero a la tabla de métodos virtuales de la clase del objeto apuntado, permite invocar al método *mVirtual1* referenciado en la tabla. Por ejemplo, la llamada anterior se transforma en algo así como:

```
(* (p->ptv[n]))(p);
```

donde $(* (p->ptv[n]))$ hace referencia al método virtual apuntado por el elemento n de la matriz de punteros a los métodos y p , que durante la ejecución apunta a un objeto concreto, es un parámetro que se añade a la lista de parámetros formales del método virtual (en el caso de *mVirtual1*, la lista de parámetros está vacía); esto es, la búsqueda de un método virtual requiere de una sobrecarga nominal durante la ejecución. El resultado es que se llama a una versión diferente del método por cada tipo de objeto.

Constructores virtuales

¿Pueden definirse constructores virtuales? Pensemos sobre ello. Un método virtual tiene que invocarse para un objeto existente, pero un constructor no, ya que su función es construir el objeto; desde este punto de vista, no tiene sentido hablar de un constructor virtual. Un método virtual es invocado a través de un puntero o referencia, pero la forma del método que se invoca será una u otra en función del tipo del objeto referenciado. En cambio, un constructor es exclusivo de un tipo exacto de objetos, otra razón más por la que no puede ser virtual. Entonces, la respuesta a la pregunta inicial es que C++ no admite constructores virtuales, pero resulta fácil simularlos. Supongamos una clase *CB*:

```
CB *p = new CB; // invoca al constructor CB
```

Esta operación podría también formar parte del cuerpo de un método que devuelva el objeto construido:

```

CB *nuevo()
{
    return new CB; // invoca al constructor CB
}

```

Si este método lo declaramos *virtual* en una clase base *CB* y lo redefinimos en una clase derivada *CD* para que devuelva un objeto *CD*, tendremos la posibilidad de crear un nuevo objeto sin conocer su tipo exacto. Por lo tanto, *nuevo* es un método que simula a un constructor virtual.

Análogamente, podemos escribir otro método virtual *clonar* que devuelva un duplicado del objeto para el cual es invocado, objeto que sabemos está referenciado por **this**. Tenemos, así, otro método que simula a un constructor copia virtual.

```

CB *clonar()
{
    return new CB(*this); // invoca al constructor copia de CB
}

```

El programa que se muestra a continuación pone en práctica lo expuesto:

```

// constructores-v.cpp
#include <iostream>
using namespace std;

```

```

class CB
{
private:
    int i;
public:
    // Constructor por omisión
    CB() { cout << "constructor CB\n"; }
    // ...
    virtual CB *nuevo() { return new CB; }
    virtual CB *clonar() { return new CB(*this); }
};

```

```

class CD : public CB
{
private:
    double d;
public:
    // Constructor por omisión
    CD() { cout << "constructor CD\n"; }
    // ...
    CD *nuevo() { return new CD; }
    CD *clonar() { return new CD(*this); }
};

```

```

CB *crearObjeto(CB *p) // función global
{
    return p->nuevo();
}

int main()
{
    CB obj_cb, *cb = &obj_cb;
    CD obj_cd, *cd = &obj_cd;

    CB *p1 = crearObjeto(cb); // crea un objeto de tipo CB
    CB *p2 = crearObjeto(cd); // crea un objeto de tipo CD

    CB *p3 = p1->clonar();
    CB *p4 = p2->clonar();

    // ...
    // Liberar memoria
    delete p1;
    delete p2;
    delete p3;
    delete p4;
}

```

Obsérvese cómo la función *crearObjeto* crea un nuevo objeto de la misma clase que el referenciado por su parámetro, y cómo el método *clonar* crea una copia del objeto referenciado por el puntero a través del que se invoca dicho método.

Se puede observar también que el tipo del objeto devuelto por los métodos *nuevo* y *clonar* de la clase derivada *CD* es *CD ** y no *CB **. Esto permite obtener un nuevo objeto sin pérdida de información de tipo. Pero, ¿no habíamos dicho que la redefinición de un método virtual en una clase derivada debía tener el mismo nombre, número y tipos de parámetros y tipo del valor retornado que en la clase base? Sí, pero sabemos que si *CB* es una clase base *pública*, cualquier función puede convertir un *CD ** a *CB **. Por lo tanto, si el tipo de retorno del método declarado virtual en la clase base es *CB **, en la redefinición de la derivada puede ser *CD **, siempre que la clase *CB* sea una clase base pública de *CD* (en algunos casos de derivación múltiple, que estudiaremos más adelante, esto puede conducir a errores durante la compilación, que no se producirán si el método virtual utiliza siempre el mismo tipo para el valor devuelto).

Destructores virtuales

Según lo estudiado hasta ahora, sabemos que cuando un objeto de una clase derivada, durante la ejecución, sale fuera del ámbito en el que ha sido definido, el objeto se destruye, lo que implica que se ejecute primero el destructor de esa clase

derivada y después el de su clase base. Ahora bien, cuando ese objeto haya sido creado dinámicamente y esté referenciado por un puntero a la clase base, pueden surgir problemas cuando requiramos su destrucción mediante el operador **delete**, porque el compilador llamará únicamente al destructor de su clase base. Esto es así porque, al ser el destructor un método no virtual, la llamada se resuelve en función del tipo del puntero.

Por ejemplo, en el programa anterior, cuando se libera la memoria asignada para cada uno de los objetos creados dinámicamente ocurre que el destructor invocado es siempre el de la clase base: `~CB`. En este caso no hay problemas, porque los destructores no tienen que realizar ninguna operación especial, como liberar recursos de memoria, por ejemplo. No obstante, ¿cuál es la solución para que los destructores se invoquen correctamente, como ocurriría con los objetos estáticos? La solución es añadir a la clase base un destructor virtual (incluso sirve un destructor vacío). Esto hace que los destructores de todas las clases derivadas sean virtuales, aunque no compartan el mismo nombre que el destructor de la clase base. De esta forma, cuando **delete** sea aplicado a un puntero a la clase base, se invocará el destructor perteneciente a la clase del objeto apuntado.

Como ejemplo, vamos a añadir un destructor virtual a la clase *CB* del programa *constructores-v.cpp*:

```
class CB
{
private:
    int i;
public:
    // Constructor por omisión
    CB() { cout << "constructor CB\n"; }
    virtual ~CB() { cout << "destructor CB\n"; }
    // ...
    virtual CB *nuevo(){ return new CB; }
    virtual CB *clonar() { return new CB(*this); }
};
```

El resultado será que al liberar la memoria asignada a los objetos creados dinámicamente:

delete p1 llamará al método *CB::~~CB*

delete p2 llamará a los métodos *CD::~~CD* y *CB::~~CB*

delete p3 llamará al método *CB::~~CB*

delete p4 llamará a los métodos *CD::~~CD* y *CB::~~CB*

Es una buena práctica dotar de un destructor virtual a una clase base que tiene métodos virtuales, aunque dicho destructor no haga nada. La razón es que una cla-

se derivada puede requerir que se defina un destructor para liberar ciertos recursos; por ejemplo, suponga que deriva una clase de *CCuenta* y que define un destructor para ella. Definiendo un destructor virtual en la clase base, se asegura que el destructor de la clase derivada será llamado cuando se necesite.

```
class CCuenta
{
    // ...
    public:
        virtual ~CCuenta() {}; // destructor
    // ...
};
```

INFORMACIÓN DE TIPOS DURANTE LA EJECUCIÓN

Según hemos visto anteriormente, cualquier operación con punteros o referencias a objetos requiere que se tenga un puntero o una referencia de un tipo apropiado para el objeto.

Operador `dynamic_cast`

El propósito del operador **`dynamic_cast`** es detectar durante la ejecución un error de conversión entre punteros o referencias que el compilador no puede determinar. En concreto nos referimos a las conversiones descendentes en una jerarquía de clases. Para que una conversión de éstas se pueda realizar con seguridad, el puntero o la referencia debe serlo a un tipo polimórfico y el objeto debe ser del tipo esperado. Esto es, para un puntero *p*, la expresión:

```
dynamic_cast<T *>(p);
```

se interpreta como la pregunta ¿es el objeto apuntado por *p* de tipo *T*? Si el objeto es de la clase *T* o de una clase derivada de una clase base *T* accesible, se trata de una conversión implícita (conversión ascendente: de derivada a base), por lo tanto, se puede prescindir del operador **`dynamic_cast`**. Por ejemplo:

```
class CB
{
    // ...
};

class CD1 : public CB
{
    // ...
};
```

```

class CD2 : protected CD1
{
    // ...
};

int main()
{
    CB *pb1 = new CB; // correcto
    // Conversiones ascendentes (de derivada a base)
    CB *pb2 = new CD1; // correcto
    CB *pb3 = new CD2; // CB es inaccesible (clase base protegida)
}

```

Si la conversión es descendente y se puede realizar, el resultado será un puntero de tipo T^* ; si no, el resultado será 0. Este tipo de conversiones se restringen a tipos polimórficos (el destino no tiene por qué ser polimórfico), porque si un objeto no posee métodos virtuales, no puede ser manipulado con seguridad si no se conoce su tipo. Por ejemplo, supongamos una clase *CB* polimórfica y una clase *CD* derivada de ella:

```

// conversiones.cpp
#include <iostream>
using namespace std;

```

```

class CB
{
    int b;
public:
    CB(int x = 10) { b = x; }
    virtual void f(){ cout << "CB\n"; }
    int obtener() { return b; }
};

```

```

class CD : public CB
{
    int d;
public:
    CD(int x = 20) { d = x; }
    void f(){ cout << "CD\n"; }
    int obtener() { return d; }
};

```

```

int main()
{
    CB* pb1 = new CB;
    CB* pb2 = new CD;

    CD* pd1 = dynamic_cast<CD*>(pb1); // pd1 = 0 (no conversión)
}

```

```

if (pd1)
{
    pd1->f();
    cout << pd1->obtener() << endl;
}

CD* pd2 = dynamic_cast<CD*>(pb2); // correcto
if (pd2)
{
    pd2->f(); // escribe CD
    cout << pd2->obtener() << endl; // escribe 20
}

delete pb1;
delete pb2;
}

```

En este ejemplo, *pd1* toma el valor 0 porque *pb1* apunta a un objeto de la clase base, no de la derivada. Si la conversión de *pb1* a *pd1* se permitiera realizar, *pd1->f()* accedería al método *CB::f* por tratarse de un método virtual, lo cual es correcto, pero *pd1->obtener()* accedería al método *CD::obtener* por tratarse de un método normal, lo cual es incorrecto porque trataría de acceder al atributo *d* que el objeto de la clase base no tiene.

Para una referencia *r*, a diferencia de los punteros, la expresión:

```
dynamic_cast<T&>(r);
```

se interpreta como la afirmación “el objeto referenciado por *r* es de tipo *T*” ya que una referencia se refiere a un objeto. Si el objeto referenciado no es del tipo esperado, **dynamic_cast** lanza una excepción **bad_cast**. Por ejemplo:

```

void fx(CB& r)
{
    CD& r1 = dynamic_cast<CD&>(r);
    // ...
}

int main()
{
    CB* pb1 = new CB;
    // ...
    try
    {
        fx(*pb1);
    }
    catch(bad_cast)
    {

```



```

    cout << "conversión entre referencias no permitida\n";
}
// ...
delete pb1;
}

```

La conversión forzada que se intenta realizar en este ejemplo lanza una excepción porque el objeto referenciado por *r* no es de la clase *CD*.

Operador typeid

Hemos visto que el operador **dynamic_cast** asegura que el código escrito funcione correctamente con clases derivadas. No obstante, en ocasiones puede ser necesario conocer el tipo exacto de un objeto. El operador **typeid** nos proporciona esta información. Su sintaxis es:

```

class type_info;
const type_info& typeid(tipo) throw();
const type_info& typeid(expresión) throw(bad_typeid);

```

Tipo es un nombre de tipo polimórfico y *expresión* es, generalmente, un objeto, un puntero o una referencia. El resultado es una referencia a un objeto de la clase **type_info** que representa el tipo de su operando. Esta clase está declarada en el fichero de cabecera `<typeinfo>` y básicamente proporciona los operadores **==** y **!=** que permiten saber si dos objetos son o no del mismo tipo, el método **name** que proporciona el nombre del tipo y el método **before** que permite la ordenación de objetos **type_info**. Si el valor del operando es 0, lanza la excepción **bad_typeid**. Por ejemplo:

```

const type_info& infol = typeid(pb1);
cout << infol.name() << endl;
cout << typeid(pd2).name() << endl;
if(infol != typeid(pd2)) cout << infol.before(typeid(pd2)) << endl;

```

Este tipo de información se debería utilizar únicamente cuando sea necesario. La verificación estática (durante la compilación) es más segura e implica menos sobrecarga. Es aconsejable utilizar los métodos virtuales en vez de la información de tipo durante la ejecución (*RTTI: runtime time identification*) cuando necesite discriminar entre tipos durante la ejecución.

POLIMORFISMO

Conseguir que los métodos de una clase base y sus redefiniciones en sus derivadas se comporten adecuadamente, independientemente del tipo del medio realmente

empleado para acceder a los mismos (puntero o referencia a una clase), se denomina *polimorfismo* (facultad de asumir muchas formas), mientras que una clase con métodos virtuales se denomina *tipo polimórfico*.

Según hemos visto, para conseguir en C++ un comportamiento polimórfico, los métodos deben ser *virtuales* y los objetos deben ser manipulados mediante punteros o referencias.

Así mismo, sabemos que una referencia a una clase derivada puede ser convertida implícitamente por C++ en una referencia a su clase base directa o indirecta. Esto significa que es posible referirse a un objeto de una clase derivada utilizando un puntero o una referencia del tipo de su clase base.

Según lo expuesto, y en un intento de buscar una codificación más genérica, pensemos ahora en una matriz de punteros en la que cada elemento señale a un objeto de alguna de las clases de la jerarquía expuesta anteriormente. ¿De qué tipo deben ser los elementos de la matriz? Según el párrafo anterior deben ser punteros o referencias a la clase *CCuenta*; de esta forma, ellos podrán almacenar indistintamente punteros o referencias a objetos de cualquiera de las clases derivadas. Por ejemplo:

```
int main()
{
    vector<CCuenta *> cuenta(3); // matriz de tres elementos de tipo
                                // punteros a CCuenta
    cuenta[0] = new CCuentaAhorro(
        "cliente01", "1234567891", 10000, 2.5, 10.0);
    cuenta[1] = new CCuentaCorriente(
        "cliente02", "2345678912", 20000, 0.5, 0.1, 6);
    cuenta[2] = new CCuentaCorrienteConIn(
        "cliente03", "3456789123", 30000, 2.0, 0.1, 6);

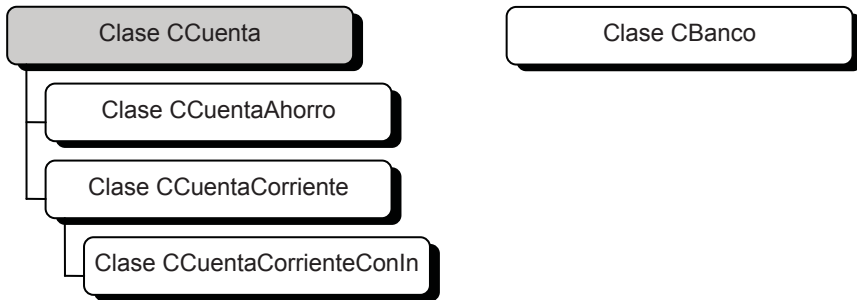
    for (int i = 0; i < 3; i++)
    {
        cout << cuenta[i]->obtenerCuenta() << endl;
        cout << cuenta[i]->intereses() << endl;
    }

    for (int i = 0; i < 3; i++)
        delete cuenta[i];
}
```

Este ejemplo define una matriz *cuenta* de tipo *CCuenta ** con tres elementos que **vector**<> inicia con el valor 0. Después crea un objeto de una de las clases derivadas y almacena su dirección en el primer elemento de la matriz; aquí C++ realizará una conversión implícita del tipo del puntero devuelto por **new** al tipo

CCuenta *. Este proceso se repetirá para cada objeto nuevo que deseemos crear (en nuestro caso tres veces). Finalmente, utilizando un bucle mostramos el número de cuenta y los intereses que se ingresarán en la misma sólo si es el primer día del mes actual. Pregunta: ¿en cuál de las dos líneas de este bucle se aplica la definición de polimorfismo? Lógicamente en la última porque, según lo estudiado hasta ahora, invoca a una u otra definición del método *intereses*, dependiendo esto del tipo del objeto para el que se ejecuta el método, utilizando el mismo medio de acceso: un puntero a *CCuenta*.

Como ejemplo, vamos a escribir un programa que cree un objeto que represente a una entidad bancaria con un cierto número de cuentas. Este objeto estará definido por una clase que denominaremos *CBanco* y las cuentas serán objetos de alguna de las clases de la jerarquía construida en los apartados anteriores.



La estructura de datos que represente el banco tiene que ser capaz de almacenar objetos *CCuentaAhorro*, *CCuentaCorriente* y *CCuentaCorrienteConIn*. Sabiendo que cualquier puntero a un objeto de una clase derivada puede convertirse implícitamente en un puntero a un objeto de su clase base, la estructura idónea es una matriz de punteros a la clase base *CCuenta*. Esta matriz será dinámica; esto es, aumentará en un elemento cuando se añada un objeto de alguna de las clases derivadas y disminuirá en uno cuando se elimine; inicialmente tendrá cero elementos. Según esto, la clase *CBanco*, que no pertenece a nuestra jerarquía, tendrá los atributos y métodos que se exponen a continuación:

Atributo	Significado
<i>cuentas</i>	Matriz de punteros de tipo <i>CCuenta</i> .
Método	Significado
<i>CBanco</i>	Es el constructor de la clase. Inicia la matriz <i>cuentas</i> con cero elementos.
<i>~CBanco</i>	Es el destructor de la clase. Libera la memoria asignada a los objetos referenciados por la matriz <i>cuentas</i> .

Método	Significado
<i>operator[]</i>	Devuelve un puntero al objeto que está en la posición <i>i</i> de la matriz <i>cuentas</i> .
<i>anyadir</i>	Añade un puntero a un objeto de la clase <i>CCuenta</i> o de alguna de sus derivadas al final de la matriz <i>cuentas</i> .
<i>eliminar</i>	Elimina el objeto que coincida con el número de cuenta pasado como argumento y después elimina el elemento de la matriz <i>cuentas</i> .
<i>buscar</i>	Devuelve la posición en la matriz <i>cuentas</i> del objeto cuyo titular (nombre total o parcial) o cuenta coincida con el valor pasado como argumento.
<i>longitud</i>	Devuelve el número de elementos de la matriz.

La definición correspondiente a esta clase se expone a continuación:

```
// banco.h - clase CBanco
#if !defined( _BANCO_H_ )
#define _BANCO_H_
#include "cuenta.h"

class CBanco
{
    // Atributos
private:
    std::vector<CCuenta *> cuentas; // matriz de objetos
// Atributos
public:
    CBanco();
    ~CBanco();
    CCuenta *operator[](unsigned int i);
    void anyadir(CCuenta *obj);
    bool eliminar(std::string cuenta);
    int buscar(std::string str, int pos);
    size_t longitud();
};

#endif // _BANCO_H_

// banco.cpp - Definición de la clase CBanco.
// Esta clase mantiene una matriz de punteros a
// objetos de cualquier tipo de cuenta bancaria.
//
#include <iostream>
#include <string>
#include <vector>
#include "banco.h"
using namespace std;
```

```
CBanco::CBanco() // constructor
{
    // Reservar espacio para 100 elementos (elementos iniciales cero)
    cuentas.reserve(100);
}

CBanco::~CBanco() // destructor
{
    // Eliminar los objetos CCuenta o de sus derivadas
    for (unsigned int i = 0; i < cuentas.size(); i++)
        delete cuentas[i];
}

CCuenta *CBanco::operator[](unsigned int i)
{
    // Devolver la referencia al objeto i de la matriz
    if (i >= 0 && i < cuentas.size())
        return cuentas[i];
    else
    {
        cout << "error: índice fuera de límites\n";
        return 0;
    }
}

void CBanco::anyadir(CCuenta *obj)
{
    // Añadir un objeto a la matriz
    cuentas.push_back(obj);
}

bool CBanco::eliminar(string cuenta)
{
    // Buscar la cuenta y eliminar el objeto
    for (unsigned int i = 0; i < cuentas.size(); i++)
        if (cuenta == cuentas[i]->obtenerCuenta())
        {
            delete cuentas[i];
            cuentas.erase(cuentas.begin()+i);
            return true;
        }
    return false;
}

int CBanco::buscar(string str, int pos)
{
    // Buscar un objeto y devolver su posición
    string nom, cuen;
    if (str.empty()) return -1;
    if (pos < 0) pos = 0;
```

```

for (unsigned int i = pos; i < cuentas.size(); i++)
{
    // Buscar por el nombre del titular de la cuenta
    nom = cuentas[i]->obtenerNombre();
    if (nom.empty()) continue;
    // ¿str está contenida en nom?
    if (nom.find(str) != string::npos)
        return i;
    // Buscar por la cuenta
    cuen = cuentas[i]->obtenerCuenta();
    if (cuen.empty()) continue;
    // ¿str es la cuenta?
    if (str == cuen)
        return i;
}
return -1;
}

size_t CBanco::longitud() { return cuentas.size(); }

```

Se puede observar que el constructor reserva un espacio inicial para 100 elementos, aunque el tamaño o número de elementos inicial es cero, y que el destructor libera la memoria asignada a los objetos referenciados por los elementos de la matriz *cuentas*; esta operación invocará al destructor de cada uno de los objetos referenciados, siempre y cuando el destructor de la clase base sea virtual.

Un usuario que escriba una aplicación que utilice esta clase y la jerarquía de clases *CCuenta* sabrá, observando la declaración del método *CBanco::anyadir*, que para añadir una cuenta tiene que pasar como argumento la dirección del objeto correspondiente a dicha cuenta. Pero, ¿qué pasaría si procede así?:

```

int main()
{
    CBanco banco;
    CCuentaCorriente cuenta01;
    banco.anyadir(&cuenta01);
}

```

Este ejemplo define dos objetos automáticos (no dinámicos): *banco* y *cuenta01*. Cuando finalice la ejecución de la función **main**, el flujo de ejecución sale fuera del ámbito de ambos objetos, lo cual hará que se invoque al destructor de *cuenta01* y después al destructor de *banco*. Esta última acción intentará, a través de la expresión *delete cuentas[i]*, eliminar otra vez el objeto *cuenta01*, que además no ha sido creado con **new**. En base a este hecho y dependiendo del compilador que utilice, el sistema operativo puede o no lanzar una excepción. Y, ¿qué pasaría si procediera de esta otra forma?:

```
int main()
{
    CBanco banco;
    CCuentaCorriente *cuenta01 = new CCuentaCorriente;
    banco.anyadir(cuenta01);
    delete cuenta01;
}
```

En este otro caso, el resultado es análogo. Cuando se ejecute *delete cuenta01* se invocará al destructor de *cuenta01*, y cuando finalice la ejecución de la función **main**, será invocado el destructor de *banco*. Esta última acción intentará, a través de la expresión *delete cuentas[i]*, eliminar otra vez el objeto *cuenta01*. En base a este hecho y dependiendo del compilador que utilice, el sistema operativo puede o no lanzar una excepción.

Este tipo de errores se puede evitar si se sigue esta pauta: “los objetos deben ser liberados por el módulo que los cree”. En el caso anterior, la clase *CBanco* ha intentado liberar un objeto que ella no ha creado, por lo que se encontró con algo que no esperaba: que el objeto ya estaba liberado. ¿Cuál es la solución? Que el método *anyadir* añada una copia del objeto creado por el usuario, de esta forma el código del usuario puede liberar sus objetos y *CBanco* los suyos. Esto implica invocar desde el método *anyadir* al constructor copia de la clase del objeto. Evidentemente, esta forma de proceder tiene también sus desventajas: duplicar objetos implica ejecutar más código, y las modificaciones que el usuario realice posteriormente tiene lógicamente que hacerlas sobre éstos últimos.

¿Cómo implementaríamos este método en la clase *CBanco*? Veamos una primera aproximación:

```
void CBanco::anyadir(CCuenta *obj)
{
    cuentas.push_back(new CCuenta(*obj));
}
```

Analizando el código anterior observamos que lo que se crea es un nuevo objeto de la clase *CCuenta*, cuando lo más seguro es que el objeto pasado como argumento sea de alguna de sus clases derivadas. Por lo tanto, lo que hay que hacer es invocar al constructor copia de su clase, dato (la clase del objeto) que no podemos conocer a través del puntero. Aunque, si utilizamos un método virtual que simule al constructor copia de la clase base y de sus derivadas, según explicamos anteriormente, el problema estará resuelto. Este método, que denominaremos *clonar*, lo declaramos virtual en la clase base *CCuenta* y lo redefiniremos en todas sus derivadas:

```
CCuenta *CCuenta::clonar() // declarado virtual
{
```

```
        return new CCuenta(*this);
    }

CCuentaAhorro *CCuentaAhorro::clonar()
{
    return new CCuentaAhorro(*this);
}

CCuentaCorriente *CCuentaCorriente::clonar()
{
    return new CCuentaCorriente(*this);
}

CCuentaCorrienteConIn *CCuentaCorrienteConIn::clonar()
{
    return new CCuentaCorrienteConIn(*this);
}
```

Una vez hayamos añadido los métodos anteriores, modificaremos el método *anyadir* así:

```
void CBanco::anyadir(CCuenta *obj)
{
    cuentas.push_back(obj->clonar());
}
```

Éste es un caso evidente de lo útil que resulta el mecanismo definido como polimorfismo. Esto es, el método *clonar* que se invoca para cada cuenta depende del tipo del objeto referenciado por *obj*.

Para finalizar, queda escribir una aplicación que utilizando la clase *CBanco* construya la entidad bancaria objetivo del ejemplo propuesto. Esta aplicación presentará un menú como el indicado a continuación:

1. Saldo
2. Buscar siguiente
3. Ingreso
4. Reintegro
5. Añadir
6. Eliminar
7. Mantenimiento
8. Salir

Opción:

La operación elegida será identificada por una sentencia **switch** y procesada de acuerdo al esquema presentado a continuación:

```
CCuenta *leerDatos(int op) { ... }
```



```
int menu() { ... }

int main()
{
    // Crear un objeto con cero elementos
    CBanco banco;

    do
    {
        opcion = menu();

        switch (opcion)
        {
            case 1: // saldo
                // Buscar un elemento por el nombre o por la cuenta.
                // La cadena de búsqueda será obtenida del teclado.
                pos = banco.buscar(cadenabuscar, 0);
                // Si se encuentra, mostrar nombre, cuenta y saldo
                break;
            case 2: // buscar siguiente
                // Buscar el siguiente elemento que contenga la cadena
                // utilizada en la última búsqueda (case 1).
                pos = banco.buscar(cadenabuscar, pos + 1);
                // Si se encuentra, mostrar nombre, cuenta y saldo
                break;
            case 3: // ingreso
            case 4: // reintegro
                // Ingresar o reintegrar una cantidad especificada.
                // Ambos datos se solicitarán del teclado.
                pos = banco.buscar(cuenta, 0);
                if (opcion == 3)
                    banco[pos]->ingreso(cantidad);
                else
                    banco[pos]->reintegro(cantidad);
                break;
            case 5: // añadir
                // Añadir un nuevo cliente. El objeto correspondiente
                // será devuelto por el método leerDatos de esta
                // aplicación, que obtendrá los datos desde el teclado.
                banco.añadir(leerDatos(tipo_objeto));
                break;
            case 6: // eliminar
                // Eliminar la cuenta especificada.
                banco.eliminar(cuenta);
                break;

            case 7: // mantenimiento
                // Cobrar comisiones e ingresar intereses, sólo
                // el día 1 de cada mes.
                for (pos = 0; pos < banco.longitud(); pos++)
                {
```

```

        banco[pos]->comisiones();
        banco[pos]->intereses();
    }
    break;
case 8: // salir
    break;
}
}
while(opción != 8);
}

```

El listado completo de la aplicación *test.cpp* se muestra a continuación. Se puede observar que utiliza cuatro funciones: *leerDato*, *leerDatos*, *menú* y **main**.

La función *leerDato* permite leer un dato real con seguridad. Esto es, cualquier entrada no válida será rechazada.

La función *leerDatos* recibe como parámetro un valor 1, 2 ó 3 dependiendo del tipo de objeto que se desee crear: *CCuentaAhorro*, *CCuentaCorriente* o *CCuentaCorrienteConIn*. Lee los atributos correspondientes al tipo de cuenta elegido e invoca al constructor adecuado. La función devuelve un puntero al nuevo objeto construido. Esta función será invocada cada vez que se elija la opción *añadir* una nueva cuenta.

La función *menú* visualiza el menú anteriormente mostrado y devuelve el entero correspondiente a la opción elegida.

La función **main** crea el objeto *banco* e invoca repetidamente a la función *menú* para permitir elegir la operación programada que se desee realizar en ese instante.

```

// test.cpp - Polimorfismo
// Aplicación para trabajar con la clase CBanco y la jerarquía
// de clases derivadas de CCuenta
//
#include <iostream>
#include <vector>
#include <limits>
#include "banco.h"
#include "cuenta_ahorro.h"
#include "cuenta_corriente.h"
#include "cuenta_corriente+.h"
using namespace std;

double leerDato()
{
    double dato = 0.0;

```

```
cin >> dato;
while (cin.fail()) // si el dato es incorrecto, limpiar el
{
    // búfer y volverlo a leer
    cout << '\a';
    cin.clear();
    cin.ignore(numeric_limits<int>::max(), '\n');
    cin >> dato;
}
// Eliminar posibles caracteres sobrantes
cin.ignore(numeric_limits<int>::max(), '\n');
return dato;
}

CCuenta *leerDatos(int op)
{
    CCuenta *obj;
    string nombre, cuenta;
    double saldo, tipoi, mant;
    cout << "Nombre.....: ";
    getline(cin, nombre);
    cout << "Cuenta.....: ";
    getline(cin, cuenta);
    cout << "Saldo.....: ";
    saldo = leerDato();
    cout << "Tipo de interés.....: ";
    tipoi = leerDato();
    if (op == 1)
    {
        cout << "Mantenimiento.....: ";
        mant = leerDato();
        obj = new CCuentaAhorro(nombre, cuenta, saldo, tipoi, mant);
    }
    else
    {
        int transex;
        double imptrans;
        cout << "Importe por transacción: ";
        imptrans = leerDato();
        cout << "Transacciones exentas...: ";
        transex = (int)leerDato();

        if (op == 2)
            obj = new CCuentaCorriente(nombre, cuenta, saldo, tipoi,
                                       imptrans, transex);
        else
            obj = new CCuentaCorrienteConIn(nombre, cuenta, saldo,
                                             tipoi, imptrans, transex);
    }
    return obj;
}
```

```
int menu()
{
    cout << "\n\n";
    cout << "1. Saldo\n";
    cout << "2. Buscar siguiente\n";
    cout << "3. Ingreso\n";
    cout << "4. Reintegro\n";
    cout << "5. Añadir\n";
    cout << "6. Eliminar\n";
    cout << "7. Mantenimiento\n";
    cout << "8. Salir\n";
    cout << endl;
    cout << "    Opción: ";
    int op;
    do
        op = (int)leerDato();
    while (op < 1 || op > 8);
    return op;
}

int main()
{
    // Crear un objeto con cero elementos
    CBanco banco;
    int opcion = 0, pos = -1;
    string cadenabuscar;
    string cuenta;
    double cantidad;
    bool eliminado = false;

    do
    {
        opcion = menu();
        switch (opcion)
        {
            case 1: // saldo
                cout << "Nombre total o parcial, o cuenta: ";
                getline(cin, cadenabuscar);
                pos = banco.buscar(cadenabuscar, 0);
                if (pos == -1)
                    if (banco.longitud() != 0)
                        cout << "búsqueda fallida\n";
                    else
                        cout << "no hay cuentas\n";
                else
                {
                    cout << banco[pos]->obtenerNombre() << endl;
                    cout << banco[pos]->obtenerCuenta() << endl;
                    cout << banco[pos]->estado() << endl;
                }
                break;
        }
    }
}
```

```
case 2: // buscar siguiente
    pos = banco.buscar(cadenabuscar, pos + 1);
    if (pos == -1)
        if (banco.longitud() != 0)
            cout << "búsqueda fallida\n";
        else
            cout << "no hay cuentas\n";
    else
    {
        cout << banco[pos]->obtenerNombre() << endl;
        cout << banco[pos]->obtenerCuenta() << endl;
        cout << banco[pos]->estado() << endl;
    }
    break;
case 3: // ingreso
case 4: // reintegro
    cout << "Cuenta: "; getline(cin, cuenta);
    pos = banco.buscar(cuenta, 0);
    if (pos == -1)
        if (banco.longitud() != 0)
            cout << "búsqueda fallida\n";
        else
            cout << "no hay cuentas\n";
    else
    {
        cout << "Cantidad: "; cantidad = leerDato();
        if (opcion == 3)
            banco[pos]->ingreso(cantidad);
        else
            banco[pos]->reintegro(cantidad);
    }
    break;
case 5: // añadir
    cout << "Tipo de cuenta < 1-(CA), 2-(CC), 3-(CCI) >: ";
    do
        opcion = (int)leerDato();
    while (opcion < 1 || opcion > 3);
    banco.anyadir(leerDatos(opcion));
    break;
case 6: // eliminar
    cout << "Cuenta: "; getline(cin, cuenta);
    eliminado = banco.eliminar(cuenta);
    if (eliminado)
        cout << "registro eliminado\n";
    else
        if (banco.longitud() != 0)
            cout << "cuenta no encontrada\n";
        else
            cout << "no hay cuentas\n";
    break;
```

```

        case 7: // mantenimiento
            for (pos = 0; pos < banco.longitud(); pos++)
            {
                banco[pos]->comisiones();
                banco[pos]->intereses();
            }
            break;
        case 8: // salir
            break;
    }
}
while(opcion != 8);
}

```

También aquí se puede observar que el mantenimiento de las cuentas (case 7) resulta sencillo gracias a la aplicación de la definición de polimorfismo. Esto es, el método *comisiones* o *intereses* que se invoca para cada cuenta depende del tipo del objeto referenciado por el elemento accedido de la matriz *cuentas* de *banco*.

Para finalizar, pensemos ahora en qué ocurriría si, por cualquier causa, tuviéramos que hacer una copia del objeto *CBanco* de la aplicación anterior; por ejemplo, porque necesitamos una copia para hacer pruebas. ¿Servirían el constructor copia y el operador de asignación que la clase implementa por omisión? La respuesta es no, porque el objeto copia tendría una matriz *cuentas* (atributo *cuentas* de *CBanco*) que haría referencia a los mismos objetos que el objeto *CBanco* original, lo que daría lugar a su destrucción cuando cualquiera de los dos objetos *CBanco* fuera eliminado. Como ejemplo, vamos a añadir al menú anterior dos opciones nuevas:

1. Saldo
2. Buscar siguiente
3. Ingreso
4. Reintegro
5. Añadir
6. Eliminar
7. Mantenimiento
8. Copia de seguridad
9. Restaurar copia de seguridad
10. Salir

Después de modificar la función *menú* para que acepte las dos opciones añadidas, modificaremos la función **main** con el fin de añadir el proceso adecuado para cada una de ellas. La opción 8 simplemente realizará un duplicado del banco; esto nos permitirá realizar pruebas con los datos actualmente en línea y cuando hayamos terminado, podemos volver al estado del banco original restaurando la copia (opción 9).

```

int main()
{
    // Crear un objeto con cero elementos
    CBanco banco;
    CBanco *copiabanco = 0; // para la copia de seguridad
    // ...
    case 8: // copia de seguridad
        if (banco.longitud() == 0) break;
        if (!copiabanco)
        {
            copiabanco = new CBanco(banco);
            if (copiabanco) cout << "copia realizada con éxito\n";
        }
        else
            cout << "existe una copia, restaurarla\n";
        break;
    case 9: // restaurar copia de seguridad
        if (!copiabanco) break;
        banco = *copiabanco;
        cout << "copia de seguridad restaurada\n";
        delete copiabanco;
        copiabanco = 0;
        break;
    case 10: // salir
        if (copiabanco) delete copiabanco;
        break;
    }
}
while(opcion != 10);
}

```

¿Cómo implementaríamos el constructor copia y el operador de asignación en la clase *CBanco*? Veámoslo a continuación:

```

// banco.h
class CBanco
{
    // ...
public:
    // ...
    CBanco(const CBanco&); // constructor copia
    CBanco& operator=(const CBanco&); // operador de asignación
    // ...
};

// banco.cpp
CBanco::CBanco(const CBanco& x)
{
    *this = x; // invoca al operador de asignación
}

```

```

CBanco& CBanco::operator=(const CBanco& x)
{
    // Eliminar las cuentas del objeto CBanco destino (*this)
    if (cuentas.size())
    {
        for (unsigned int i = 0; i < cuentas.size(); i++)
            delete cuentas[i];
        // Eliminar todos los elementos de la matriz cuentas
        cuentas.clear();
    }
    // Copiar el banco origen, x, en el banco destino
    for (unsigned int i = 0; i < x.cuentas.size(); i++)
        cuentas.push_back(x.cuentas[i]);
    return *this;
}

```

Puesto que tanto el constructor copia como el operador de asignación tienen como objetivo copiar un objeto en otro, el código involucrado en la copia es el mismo, por eso el constructor copia invoca al operador de asignación. Algo que necesita hacer el operador de asignación, que no hace el constructor copia, es borrar la matriz de objetos destino reduciéndola a una matriz con cero elementos, antes de realizar la copia. Observemos ahora cómo se copia el objeto origen en el destino:

```

for (unsigned int i = 0; i < x.cuentas.size(); i++)
    cuentas.push_back(x.cuentas[i]);

```

La copia que realiza este código no es correcta. ¿Por qué? Porque estamos copiando las direcciones de los objetos referenciados (*x.cuentas[i]* es de tipo *CCuenta **) y no estamos duplicando los objetos; al final, ambas matrices, origen y destino, estarán referenciando a los mismos objetos, problema que ya abordamos en el capítulo *Clases*. Lo que tenemos que hacer entonces es duplicar el objeto apuntado por *x.cuentas[i]* invocando al método virtual *clonar* definido en la clase base *CCuenta* y redefinido en todas sus derivadas. De acuerdo con esto, modificaremos el código que realiza la copia del objeto origen en el destino así:

```

for (unsigned int i = 0; i < x.cuentas.size(); i++)
    cuentas.push_back(x.cuentas[i]->clonar());

```

CLASES ABSTRACTAS

Pensemos en un objeto genérico, tal como *figura*, del cual utilizaremos variantes concretas, como *círculo*, *cuadrado* y *triángulo*. Pensando así, no tiene sentido crear objetos *figura*, pero sí lo tiene crear cuadrados y triángulos. Siguiendo en esta línea, ¿de qué clase eran las cuentas que creamos para los clientes del banco en el ejemplo anterior? (el realizado en el apartado *Polimorfismo*). Eran de las clases

CCuentaAhorro, *CCuentaCorriente* y *CCuentaCorrienteConIn*. Entonces, ¿cuál era la misión de la clase *CCuenta*? Actuar como base de sus clases derivadas, agrupando las operaciones comunes a todas ellas. Como vemos, algunas clases, como *CCuenta*, representan objetos abstractos, análogos a *figura*, por lo que no tiene sentido crear objetos de ellas, aunque sí lo tiene crearlos de sus derivadas, de ahí que reciban el nombre de clases abstractas.

Según lo expuesto, definiremos una *clase abstracta* como una clase que puede utilizarse solamente como clase base de otras clases. Y, ¿cómo indicamos a C++ que una clase es abstracta? Añadiéndola un método *virtual puro*, que es un método virtual con el iniciador “= 0”.

Según esto, vamos a modificar la aplicación anterior para declarar la clase *CCuenta* abstracta. La clase *CCuenta* tiene tres métodos virtuales: *comisiones*, *intereses* y *clonar*. Los dos primeros ya vimos que no hacían nada, por lo tanto son candidatos a ser métodos virtuales puros, y el tercero devolvía un nuevo objeto *CCuenta*, pero como hemos llegado a la conclusión de que esta clase cumple los requisitos para ser abstracta, no se podrán crear objetos de ella, por lo tanto, este método también lo declararemos virtual puro. Para ello, eliminaremos las definiciones de cada uno de los métodos y escribiremos la declaración de la clase así:

```
class CCuenta
{
    // Atributos
private:
    string nombre;
    string cuenta;
    double saldo;
    double tipoDeInteres;

    // Métodos
public:
    CCuenta(string nom = "sin nombre", string cue = "0000",
            double sal = 0.0, double tipo = 0.0);
    CCuenta(const CCuenta&);
    virtual ~CCuenta() {}; // destructor
    CCuenta& operator=(const CCuenta&);
    bool asignarNombre(string nom);
    string obtenerNombre();
    bool asignarCuenta(string cue);
    string obtenerCuenta();
    double estado();
    virtual void comisiones() = 0;
    virtual double intereses() = 0;
    bool ingreso(double cantidad);
    void reintegro(double cantidad);
    double obtenerTipoDeInteres();
};
```

```

    bool asignarTipoDeInteres(double tipo);
    virtual CCuenta *clonar() = 0;
};

```

El hecho de que hayamos eliminado las definiciones de los métodos virtuales puros no quiere decir que un método virtual puro no pueda tener su definición; puede tenerla igual que cualquier otro método, aunque no sea muy común. En este caso, para invocarlo, a través de un objeto de una clase derivada, habría que calificarlo con el nombre de la clase utilizando el operador de ámbito.

A partir de la declaración anterior de *CCuenta*, ésta pasa a ser una clase abstracta y sólo puede utilizarse como interfaz y como base para otras clases.

Un método virtual puro se hereda como tal. Por esta razón, una clase derivada debe proporcionar una redefinición para cada uno de los métodos virtuales de su clase base, ya que de no hacerlo, los heredará y se convertirá en una clase abstracta, lo que no permitirá declarar objetos de la misma. En nuestro ejemplo, las clases *CCuentaAhorro*, *CCuentaCorriente* y *CCuentaCorrienteConIn* redefinen todas los tres métodos virtuales puros: *comisiones*, *intereses* y *clonar*.

Resumiendo, no se pueden crear objetos de una clase abstracta. De intentarlo, el compilador mostrará un error. Por ejemplo:

```
CCuenta cuenta01; // Error: clase abstracta
```

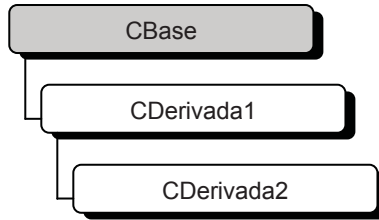
Tampoco se puede utilizar una clase abstracta como tipo en la lista de parámetros de una función (piense que los parámetros de una función son creados en el instante en el que ésta se invoca), como tipo devuelto por una función o como tipo en una conversión explícita. Sí es posible declarar punteros y referencias a una clase abstracta. Por ejemplo:

```
CCuenta *p; // correcto
CCuenta& fn(CCuenta &); // correcto
```

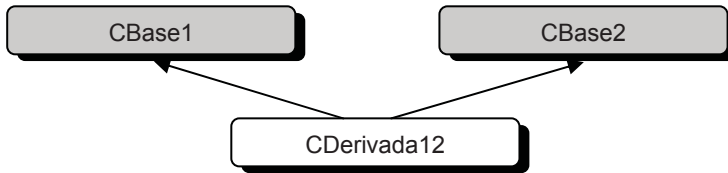
HERENCIA MÚLTIPLE

Una clase derivada puede tener una o más clases base directas. Cuando una clase derivada tiene una sola clase base directa, estamos en el caso de *herencia simple*.

En este caso, el objeto de la clase derivada se compone de los miembros heredados de la clase base y de los propios de la clase derivada.



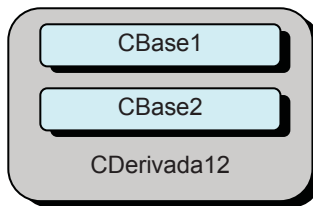
Por el contrario, cuando una clase tiene más de una clase base directa, según muestra el gráfico siguiente, estamos en el caso de *herencia múltiple*:



Esta jerarquía de clases vista desde C++ se implementaría así:

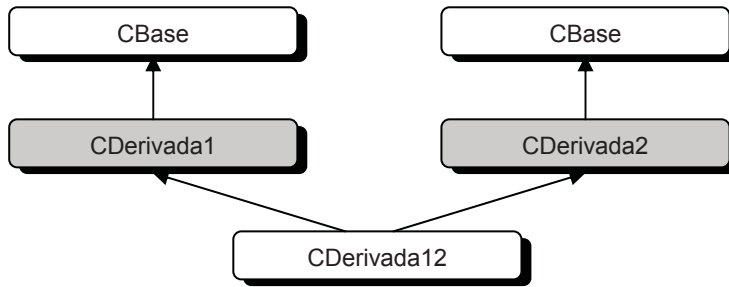
```
class CDerivada12 : public CBase1, public CBase2 { ... }
```

En este caso, un objeto de *CDerivada12* estará formado por los miembros heredados de *CBase1*, más los miembros heredados de *CBase2*, más sus propios miembros. La herencia múltiple permite que clases hermanas (*CBase1* y *CBase2* se denominan clases hermanas respecto de *CDerivada12*) compartan información sin que exista una dependencia con respecto a una clase base común.



El acceso a un miembro de *CBase1*, de *CBase2* o de *CDerivada12* se hace exactamente igual que en la herencia simple. En cuanto a los métodos virtuales, trabajan de la forma acostumbrada, no obstante, al final de este apartado haremos alguna observación para tener en cuenta.

Por otra parte, el hecho de que podamos especificar más de una clase base implica la posibilidad de tener otra clase base dos veces (nos referimos a la clase base de éstas que actúan ahora como base en la derivación múltiple); en este caso se dice que la clase base está replicada. Por ejemplo:



En este ejemplo, tanto la clase *CDerivada1* como *CDerivada2* se han derivado de la misma clase, *CBase*, por lo que *CDerivada12* heredará tanto los miembros de *CDerivada1* como de *CDerivada2*, lo que implica heredar dos veces los miembros de *CBase*.

Un ejemplo práctico podría ser una jerarquía de clases que permita crear términos de un polinomio dependiente de dos variables x e y . Por ejemplo:

$$7x^3y^2 - 4x^2 + 2y - 5$$

La clase base de esta jerarquía podría ser *CTermino* con un dato miembro que almacene el coeficiente de un término cualquiera. De ella se derivarían *CTerminoEnX*, con un dato miembro que almacene el exponente de x , y *CTerminoEnY*, con un dato miembro que almacene el exponente de y , y de éstas dos se derivaría *CTerminoEnXY*. De esta forma, un objeto de la clase *CTermino* puede representar a un término independiente, un objeto de la clase *CTerminoEnX* puede representar a un término en x , un objeto de la clase *CTerminoEnY* puede representar a un término en y , y un objeto de la clase *CTerminoEnXY* puede representar a un término en xy .

Si ahora suponemos que inicialmente ya existían las clases *CTerminoEnX* y *CTerminoEnY*, cabe destacar el papel tan importante que hace la herencia múltiple en la fusión de clases existentes. En definitiva, ésta es la aplicación más común de la herencia múltiple.

Apoyándonos en este ejemplo práctico, vamos a implementar una jerarquía de clases igual a la mostrada en la figura anterior:

```

class CTermino // clase base
{
private:
    double coeficiente;
public:
    CTermino(double k = 1) : coeficiente(k) {}
    double coef() { return coeficiente; }
};
  
```

```

class CTerminoEnX : public CTermino
{
private:
    int exponenteDeX;
public:
    CTerminoEnX(double k = 1, int e = 0) :
        CTermino(k), exponenteDeX(e) {}
    int expX() { return exponenteDeX; }
    void mostrarTx() { cout << coef() << "x^" << exponenteDeX; }
};

class CTerminoEnY : public CTermino
{
private:
    int exponenteDeY;
public:
    CTerminoEnY(double k = 1, int e = 0) :
        CTermino(k), exponenteDeY(e) {}
    int expY() { return exponenteDeY; }
};

class CTerminoEnXY : public CTerminoEnX, public CTerminoEnY
{
public:
    CTerminoEnXY(double k = 1, int ex = 0, int ey = 0) :
        CTerminoEnX(k, ex), CTerminoEnY(k, ey) {}
    void mostrarTxy() {
        cout << coef() << "x^" << expX() << "y^" << expY(); }
};

```

El hecho de que cada objeto de *CTerminoEnXY* contenga dos copias de los miembros de *CTermino* implica que las referencias a los mismos producirán errores de ambigüedad. Por ejemplo, la siguiente sentencia produce un error de ambigüedad:

```
cout << coef() << "x^" << expX() << "y^" << expY(); }
```

Lo que sucede es que el compilador no puede decidir qué copia de *coef* tiene que utilizar: la copia heredada de *CTermino* a través de *CTerminoEnX* o la heredada a través de *CTerminoEnY*. Una solución es especificar explícitamente la copia a utilizar. Por ejemplo:

```
cout << CTerminoEnX::coef() << "x^" << expX() << "y^" << expY();
```

Otros problemas de ambigüedad pueden surgir por otras causas y generalmente la solución será utilizar el operador de resolución del ámbito, o simplemente ejecutar una conversión *cast*. Por ejemplo, suponga que deseamos convertir un

puntero a un objeto de la clase *CTerminoEnXY* en un puntero a la clase *CTermino*. Si procedemos como se indica a continuación, se producirá un error:

```
CTermino *pCTermino = 0;
CTerminoEnXY *pCTerminoEnXY = 0;
// ...
pCTermino = pCTerminoEnXY; // error: ambigüedad
```

Una vez más, el compilador no puede decidir si la conversión la realiza a través de *CTerminoEnX* o a través de *CTerminoEnY*. Para deshacer la ambigüedad, tendremos que utilizar una conversión *cast*, por ejemplo, así:

```
pCTermino = static_cast<CTerminoEnX *>(pCTerminoEnXY);
```

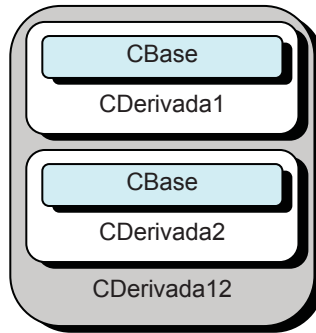
Obsérvese que primero se convierte explícitamente *pCTerminoEnXY* en un puntero a *CTerminoEnX*, y después éste se convierte implícitamente en un puntero a *CTermino*.

Lo expuesto hasta ahora demuestra que una clase base que se replique igual que *CTermino* no se debería utilizar fuera de sus clases derivadas inmediatas (por ejemplo, *CTermino* no debería utilizarse fuera de *CTerminoEnX* o de *CTerminoEnY*), y cuando se haga desde cualquier punto donde se vea más de una copia de la base, los accesos a la funcionalidad de la misma deben calificarse explícitamente para resolver la ambigüedad.

Clases base virtuales

En la jerarquía de clases presentada anteriormente, el mecanismo de herencia normal hace que la clase derivada *CDerivada12* herede dos veces los miembros de la clase *CBase*: una a través de la clase *CDerivada1* y otra a través de la clase *CDerivada2*. Esto implica que cuando creamos un objeto *CDerivada12*, éste contendrá dos subobjetos de *CBase* (subobjetos: áreas de memoria para soportar la clase base), lo que origina, no sólo los errores de ambigüedad comentados anteriormente, sino un derroche de espacio. De acuerdo con lo expuesto, podemos imaginarnos un objeto *CDerivada12* según muestra la figura siguiente.

Para evitar que la clase base común se replique en la clase más derivada (*CBase* se replica en *CDerivada12*), es necesario poner en marcha el mecanismo de herencia virtual. Esto hará que la clase base común pase de ser clase base replicada a clase base virtual.



¿Cómo se activa el mecanismo de herencia virtual? Este mecanismo se pone en marcha cuando, en un proceso de derivación, la clase base común se declara **virtual**, lo que asegurará que sólo se utilizará un subobjeto de la misma.

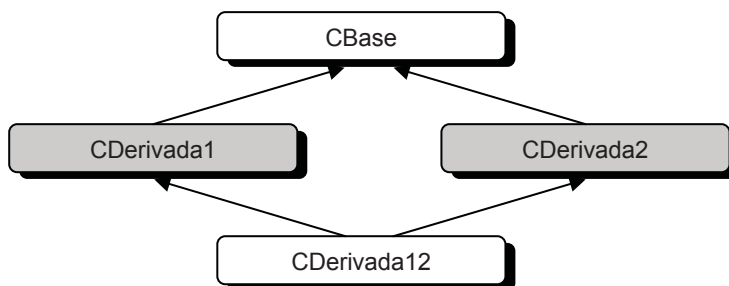
Por ejemplo, para evitar que *CBase* se replique en la clase *CDerivada12*, en el proceso de derivación la declararemos **virtual**, así:

```
class CBase // clase base
{
    // Miembros de la clase
};

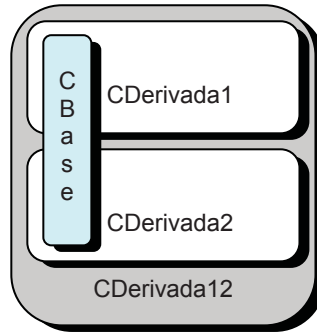
class CDerivada1 : public virtual CBase
{
    // Miembros de la clase
};

class CDerivada2 : public virtual CBase
{
    // Miembros de la clase
};

class CDerivada12 : public CDerivada1, public CDerivada2
{
    // Miembros de la clase
};
```



En un proceso de herencia múltiple, toda clase base que se declare virtual será representada en las derivadas mediante un solo subobjeto, y cada clase no declarada virtual tendrá su propio subobjeto:



Cuando se construye un objeto completo (un objeto de la clase más derivada), el constructor de una *clase base virtual* es siempre llamado antes que los constructores de las clases base no virtuales, independientemente del orden en el que se hayan especificado en la lista de iniciación. Esto es, cualquier subobjeto virtual se inicia antes que otro no virtual. Los destructores son siempre invocados en el orden inverso.

La siguiente versión del programa anterior demuestra con claridad cómo afrontar un caso como el expuesto.

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

class CTermino // clase base
{
private:
    double coeficiente;
public:
    CTermino(double k = 1) : coeficiente(k) {}
    double coef() { return coeficiente; }
};

class CTerminoEnX : public virtual CTermino
{
    // ...
};

class CTerminoEnY : public virtual CTermino
{
    // ...
};
```



```

class CTerminoEnXY : public CTerminoEnX, public CTerminoEnY
{
public:
    CTerminoEnXY(double k = 1, int ex = 0, int ey = 0) :
        CTermino(k), CTerminoEnX(k, ex), CTerminoEnY(k, ey) {}
    void mostrarTxy() {
        cout << coef() << "x^" << expX() << "y^" << expY(); }
};

int main()
{
    CTerminoEnXY *pCTerminoEnXY = new CTerminoEnXY(2, 5, 3);
    pCTerminoEnXY->mostrarTxy();
    delete pCTerminoEnXY;
}

```

Obsérvese que al derivar las clases *CTerminoEnX* y *CTerminoEnY*, se declara la clase *CTermino* **virtual**. Esto asegura que el compilador solamente pasará a la clase *CTerminoEnXY* una copia de *CTermino*.

Veamos ahora cómo se construye un objeto completo. Por ejemplo, cuando se ejecuta la sentencia:

```
CTerminoEnXY *pCTerminoEnXY = new CTerminoEnXY(2, 5, 3);
```

se invoca al constructor de *CTerminoEnXY* para construir un objeto dinámico de esta clase, el cual llama para su ejecución primero al constructor de *CTermino*, después al constructor de *CTerminoEnX*, después al constructor de *CTerminoEnY* y por último se ejecuta el cuerpo del constructor *CTerminoEnXY*; esto es, el constructor de la *clase base virtual* es llamado antes que los constructores de las clases base no virtuales. Los destructores son ejecutados en el orden inverso. Como consecuencia, en la lista de iniciadores del constructor *CTerminoEnXY* se ha especificado *CTermino(k)*; de no hacerlo así, *coeficiente* sería iniciado con el valor por omisión.

Para poder observar el orden de ejecución de los constructores y de los destructores (ojo, no confunda el orden en el que se llaman los constructores con el orden en el que se ejecuta el cuerpo de cada uno de ellos), puede hacer que éstos muestren un mensaje cuando se ejecuten. En este caso, si ejecuta el código de la función **main** anterior, podrá observar una solución similar a la mostrada a continuación:

```

constructor de CTermino
constructor de CTerminoEnX
constructor de CTerminoEnY
constructor de CTerminoEnXY
2x^5y^3

```

```

destructor de CTerminoEnXY
destructor de CTerminoEnY
destructor de CTerminoEnX
destructor de CTermino

```

Si la clase base no fuera virtual, el iniciador $CTermino(k)$ del constructor $CTerminoEnXY$ no tendría sentido, porque $CTermino$ no es una clase base directa. En este caso, resolviendo los problemas de ambigüedad, el resultado de ejecutar el código anterior sería el siguiente:

```

constructor de CTermino
constructor de CTerminoEnX
constructor de CTermino
constructor de CTerminoEnY
constructor de CTerminoEnXY
 $2x^5y^3$ 
destructor de CTerminoEnXY
destructor de CTerminoEnY
destructor de CTermino
destructor de CTerminoEnX
destructor de CTermino

```

Redefinición de métodos de bases virtuales

Un método virtual de una clase base virtual puede ser redefinido en una clase derivada directa o indirecta. Por ejemplo, volviendo al esquema general anterior, si $CBase$ declara un método virtual, éste puede ser redefinido por $CDerivada1$, por $CDerivada2$ o por $CDerivada12$.

Cuando la interfaz de una clase base virtual proporciona varios métodos virtuales, no todos tienen que ser redefinidos por todas las clases derivadas, sino que clases derivadas diferentes pueden redefinir métodos diferentes. Por ejemplo, supongamos que $CBase$ declara los métodos virtuales $mvb1$ y $mvb2$; $CDerivada1$, podría redefinir $mvb1$ y $CDerivada2$ $mvb2$.

¿Qué sucede si clases derivadas diferentes redefinen el mismo método? Esto es válido si y sólo si cada una de las clases queda bien formada. En una clase bien formada, por cada método virtual declarado en esa clase o en cualquiera de sus clases base directas o indirectas, hay un único método que redefine ese método en cada una de las otras clases que lo redefinen.

Por ejemplo, si $CBase$ declara un método virtual mvb , que es redefinido por $CDerivada1$ y $CDerivada2$, sería un error que no fuera redefinido también por $CDerivada12$, ya que, por haber heredado esta clase mvb dos veces, no habría en ella un único método que redefine mvb en cada una de las otras clases base direc-

tas o indirectas que lo redefinen. A continuación se muestra otra versión del programa anterior con la que puede experimentar lo expuesto:

```
#include <iostream>
using namespace std;
class CTermino // clase base
{
private:
    double coeficiente;
public:
    // Constructor
    CTermino(double k = 1) : coeficiente(k) {}
    double coef() { return coeficiente; }
    virtual void mostrar() = 0;
};

class CTerminoEnX : public virtual CTermino
{
private:
    int exponenteDeX;
public:
    // Constructor
    CTerminoEnX(double k = 1, int e = 0) :
        CTermino(k), exponenteDeX(e) {}
    int expX() { return exponenteDeX; }
    void mostrar() { cout << coef() << "x^" << exponenteDeX; }
};

class CTerminoEnY : public virtual CTermino
{
private:
    int exponenteDeY;
public:
    // Constructor
    CTerminoEnY(double k = 1, int e = 0) :
        CTermino(k), exponenteDeY(e) {}
    int expY() { return exponenteDeY; }
};

class CTerminoEnXY : public CTerminoEnX, public CTerminoEnY
{
public:
    // Constructor
    CTerminoEnXY(double k = 1, int ex = 0, int ey = 0) :
        CTermino(k), CTerminoEnX(k, ex), CTerminoEnY(k, ey) {}
    void mostrar() { cout << coef() << "x^" << expX()
        << "y^" << expY(); }
};
```

```

ostream& operator<<(ostream& os, CTermino* t)
{
    t->mostrar();
}

int main()
{
    CTerminoEnX *pCTerminoEnX = new CTerminoEnX(3, 2);
    cout << pCTerminoEnX << endl;
    CTerminoEnXY *pCTerminoEnXY = new CTerminoEnXY(2, 5, 3);
    cout << pCTerminoEnXY << endl;
    delete pCTerminoEnX;
    delete pCTerminoEnXY;
    cout << endl;
}

```

Este ejemplo constituye una jerarquía de clases con una clase base virtual polimórfica *CTermino*, de la que se derivan dos clases hermanas, *CTerminoEnX* y *CTerminoEnY*, de las que se deriva otra clase *CTerminoEnXY*. La clase *CTermino* se ha declarado abstracta incluyendo el método virtual puro *mostrar*, y si analizamos sus clases derivadas directas o indirectas, todas están bien formadas. La clase *CTerminoEnY* también es abstracta porque no redefine el método *mostrar*. Esto se ha hecho así porque para crear términos dependientes de una variable ya tenemos la clase *CTerminoEnX*.

Conversiones entre clases

Cuando trabajamos con una jerarquía de clases, según los requerimientos del desarrollo que estemos realizando, pueden surgir conversiones descendentes (de clase base a derivada), conversiones ascendentes (de clase derivada a base) o conversiones cruzadas (de clase hermana a hermana). Por ejemplo, si en la jerarquía de clases del ejemplo anterior necesitáramos realizar algunas de estas conversiones, ¿cómo se podrían realizar? Veámoslo con un ejemplo:

```

1. void f(CTerminoEnXY* p)
2. {
3.     // Clase base normal
4.     CTerminoEnX *pTx = p; // apunta a un objeto de CTerminoEnXY
5.     CTerminoEnXY *pTxy = static_cast<CTerminoEnXY *>(pTx);
6.     pTxy = dynamic_cast<CTerminoEnXY *>(pTx);
7.
8.     // Clases hermanas
9.     CTerminoEnY *pTy = dynamic_cast<CTerminoEnY *>(pTx);
10.    pTy = static_cast<CTerminoEnY *>(pTx); // Error
11.
12.    // Clase base virtual
13.    CTermino *pT = p; // apunta a un objeto de CTerminoEnXY

```

```

14. pTxy = static_cast<CTerminoEnXY *>(pT); // Error
15. pTxy = dynamic_cast<CTerminoEnXY *>(pT);
16. }

```

La línea 4 realiza una conversión implícita de derivada a base normal, la línea 5 realiza una conversión sin verificación de base normal a derivada y la línea 6 realiza una conversión con verificación de base normal a derivada. Esto pone de manifiesto que las conversiones ascendentes, cuando la clase base es normal, no requieren de ningún operador y las descendentes pueden realizarse con o sin verificación del tipo del objeto.

Las líneas 9 y 10 realizan una conversión con verificación y sin ella, respectivamente, de hermana a hermana. La primera da lugar a un error durante la compilación, lo que demuestra que este tipo de conversiones requieren verificación del tipo del objeto.

La línea 13 realiza una conversión implícita de derivada a base virtual, la línea 14 realiza una conversión sin verificación de base virtual a derivada y la línea 15 realiza una conversión con verificación de base virtual a derivada. Esto demuestra que las conversiones ascendentes, cuando la clase base es virtual, no requieren de ningún operador y las descendentes sólo pueden realizarse con verificación del tipo del objeto.

Como conclusión, un operador **dynamic_cast** puede realizar conversiones de una clase virtual polimórfica a una clase derivada o entre hermanas y el operador **static_cast** no, porque no verifica el tipo del objeto. Obsérvese que **dynamic_cast** requiere un operando polimórfico para encontrar el objeto; un objeto no polimórfico no almacena ninguna información, por eso no se puede utilizar cuando se necesita recorrer una jerarquía de clases para encontrar una apropiada para usarla como interfaz.

EJERCICIOS RESUELTOS

1. Partiendo de las clases *CEstudios*, *CAlumno*, *CAsignatura*, *CConvocatoria* y *CFecha* construidas en el capítulo anterior (apartado *Ejercicios propuestos*), vamos a añadir dos nuevas clases *CAsignaturaOb* y *CAsignaturaOp* derivadas de *CAsignatura* con el fin de poder disponer de una lista de asignaturas obligatorias y de otra de optativas. Con este nuevo diseño tenemos que pensar en que ahora la matriz *CAlumno::asignatura* de tipo *CAsignatura ** hará referencia a objetos *CAsignaturaOb* y *CAsignaturaOp*. Entonces, cuando necesitemos duplicar un objeto *CAlumno*, tendremos que duplicar los objetos *CAsignaturaOb* y *CAsignaturaOp* apuntados por la matriz *asignatura*. Esto requerirá conocer el tipo de los objetos, para lo que necesitaremos trabajar con clases polimórficas. Esto es, las clases *CA-*

signaturaOb y *CAsignaturaOp* redefinirán un método *clonar* (que simulará al constructor copia de su clase) declarado virtual en la clase base; su misión será duplicar un objeto basándose en su tipo, no en el puntero que lo referencia.

Según lo expuesto, la clase *CAsignatura* se modificaría de la forma siguiente:

```
class CAsignatura
{
    // ...

public:
    // ...

    virtual ~CAsignatura(){}
    virtual CAsignatura *clonar() = 0;
};
```

El método *clonar* se ha declarado virtual puro con el fin de declarar abstracta la clase *CAsignatura*, ya que ahora no tiene sentido crear objetos de este tipo. También se ha declarado virtual el destructor; de esta forma, cuando se destruyan los objetos referenciados por la matriz *asignatura*, será invocado el destructor de la clase del objeto que, a su vez, invocará al destructor de su clase base. En este caso, de no proceder de la forma expuesta, todo funcionaría correctamente, puesto que los destructores de *CAsignaturaOb* y *CAsignaturaOp* no tienen nada que hacer que no sea hecho por los destructores por omisión, pero obrar de esta forma supone un buen estilo de programación. Por eso, las clases *CAsignaturaOb* y *CAsignaturaOp* sólo redefinirán el método *clonar*.

```
// asignaturaOb.h - Declaración de la clase CAsignaturaOb
#ifndef _ASIGNATURAOB_H_
#define _ASIGNATURAOB_H_
#include "asignatura.h"
```

```
class CAsignaturaOb : public CAsignatura
{
public:
    CAsignaturaOb(int id = 999999, string nom = "");
    CAsignaturaOb *clonar();
};
```

```
#endif // _ASIGNATURAOB_H_
```

```
// asignaturaOb.cpp - Definición de la clase CAsignaturaOb
#include <iostream>
#include "asignaturaob.h"
```

```
CAsignaturaOb::CAsignaturaOb(int id, string nom) :
    CAsignatura(id, nom) {}
```

```

CAsignaturaOb *CAsignaturaOb::clonar()
{
    return new CAsignaturaOb(*this);
}

// asignaturaOp.h - Declaración de la clase CAsignaturaOp
#ifndef _ASIGNATURAOP_H_
#define _ASIGNATURAOP_H_
#include "asignatura.h"

class CAsignaturaOp : public CAsignatura
{
public:
    CAsignaturaOp(int id = 999999, string nom = "");
    CAsignaturaOp *clonar();
};

#endif // _ASIGNATURAOP_H_

```

```

// asignaturaOp.cpp - Definición de la clase CAsignaturaOp
#include <iostream>
#include "asignaturaop.h"

CAsignaturaOp::CAsignaturaOp(int id, string nom) :
    CAsignatura(id, nom) {}

CAsignaturaOp *CAsignaturaOp::clonar()
{
    return new CAsignaturaOp(*this);
}

```

Modificamos a continuación el constructor copia y el operador de asignación de la clase *CAlumno*:

```

CAlumno::CAlumno(const CAlumno& x)
{
    *this = x;
}

CAlumno& CAlumno::operator=(const CAlumno& x)
{
    // Eliminar las asignaturas del objeto CAlumno destino (*this)
    for (unsigned int i = 0; i < asignatura.size(); i++)
        delete asignatura[i];
    // Redimensionar la matriz asignatura del destino (*this)
    asignatura.resize(x.asignatura.size());

    // Copiar el alumno origen, x, en el alumno destino
    DNI = x.DNI;
    nombre = x.nombre;
}

```

```

    direccion = x.direccion;
    for (unsigned int i = 0; i < x.asignatura.size(); i++)
        asignatura[i] = x.asignatura[i]->clonar();

    return *this;
}

```

A continuación, vamos a sobrecargar el operador de inserción para que permita añadir una o más asignaturas a la lista de asignaturas de un alumno; esto es, se trata de permitir operaciones como ésta: *alumno01 << pAsig01 << pAsig02* (*pAsig01* y *pAsig02* son de tipo *CAsignatura **).

```

CAlumno& CAlumno::operator<<(CAsignatura *asig)
{
    asignatura.push_back(asig->clonar());
    return *this;
}

```

Obsérvese que en la matriz *asignatura* no almacenamos el puntero pasado como argumento, sino que creamos un nuevo objeto invocando al método *clonar* y almacenamos el puntero al objeto devuelto por ésta. Esto permitirá seguir la regla de que “los objetos deben ser liberados por el módulo que los cree”, según se explicó anteriormente en este mismo capítulo.

2. Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de las variables x e y . Por ejemplo:

$$2x^3y - xy^3 + 8.25 \text{ más } 5x^5y - 2x^3y + 7x^2 - 3 \text{ igual a } 5x^5y + 7x^2 - xy^3 + 5.25$$

Cada término del polinomio será representado por una clase *CTermino* y cada polinomio por una clase *CPolinomio*. La declaración de la clase *CTermino* se guardará en un fichero *termino.h* y su implementación en *termino.cpp*, y *CPolinomio* en los ficheros *polinomio.h* y *polinom.cpp*.

Antes de empezar con este ejercicio, es conveniente que repase el trabajo que realizó cuando resolvió los ejercicios propuestos en los dos capítulos anteriores. Si no lo hizo, es aconsejable que lo haga antes de empezar a estudiar éste.

Quizás se pregunte qué sentido tiene realizar este ejercicio si no trata con clases derivadas. La respuesta es sencilla: este ejercicio es la antesala a uno de los ejercicios que a continuación se proponen.

La clase *CTermino* puede escribirse así:

```

// termino.h - Declaración de la clase CTermino
//

```



```

#if !defined( _TERMINO_H_ )
#define _TERMINO_H_

/////////////////////////////////////////////////////////////////
// Clase para manipular un término de un polinomio dependiente
// de las variables x e y.
class CTermino
{
protected:
    float Coeficiente; // coeficiente
    int ExponenteDeX; // exponente de x
    int ExponenteDeY; // exponente de y
public:
    CTermino(float coef = 0.0, int expx = 1, int expy = 1);
    void AsignarCoeficiente(float);
    float ObtenerCoeficiente() const;
    void AsignarExponenteDeX(int);
    int ObtenerExponenteDeX() const;
    void AsignarExponenteDeY(int);
    int ObtenerExponenteDeY() const;
    void VisualizarTermino();
};
/////////////////////////////////////////////////////////////////

#endif // _TERMINO_H_

// termino.cpp - Implementación de la clase CTermino
//
#include <iostream>
#include <cmath>
#include "termino.h" // clases CTermino
using namespace std;

/////////////////////////////////////////////////////////////////
CTermino::CTermino(float coef, int expx, int expy) :
Coeficiente(coef), ExponenteDeX(expx), ExponenteDeY(expy) {}

void CTermino::AsignarCoeficiente(float coef)
{
    Coeficiente = coef;
}

float CTermino::ObtenerCoeficiente() const
{
    return Coeficiente;
}

void CTermino::AsignarExponenteDeX(int exp)
{
    ExponenteDeX = exp;
}

```

```

int CTermino::ObtenerExponenteDeX() const
{
    return ExponenteDeX;
}

void CTermino::AsignarExponenteDeY(int exp)
{
    ExponenteDeY = exp;
}

int CTermino::ObtenerExponenteDeY() const
{
    return ExponenteDeY;
}

// Visualizar un término
void CTermino::VisualizarTermino()
{
    cout << ((Coeficiente < 0) ? " - " : " + ")
         << fabs(Coeficiente);
    if (ExponenteDeX) cout << "x^" << ExponenteDeX;
    if (ExponenteDeY) cout << "y^" << ExponenteDeY;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

La clase *CTermino* representa un término del polinomio, el cual queda perfectamente definido cuando se conoce su coeficiente, el exponente de la variable x y el exponente de la variable y : *coeficiente*, *ExponenteDeX* y *ExponenteDeY*.

Para acceder a los atributos de un término se han implementado los métodos típicos de asignar y obtener el valor almacenado en el atributo que se trate en cada caso. Otros métodos implementados son: un constructor con argumentos con valores por omisión para permitir construir un objeto *CTermino* a partir de unos valores determinados y un método *VisualizarTermino* para mostrar un término en la pantalla.

Es evidente que extender esta clase a ecuaciones dependientes de más de dos variables no entraña ninguna dificultad; es cuestión de añadir más datos miembro y los métodos de acceso correspondientes.

Siguiendo con el desarrollo, la clase *CPolinomio* puede escribirse así:

```

// polinomio.h - Declaración de la clase CPolinomio
//
#ifdef _CPOLINOMIO_H_
#define _CPOLINOMIO_H_

#include <vector>

```

```

#include "termino.h" // clases CTermino y CPolinomio
using namespace std;

////////////////////////////////////
// Clase para manipular ecuaciones algebraicas o polinómicas
// dependientes de dos variables.
class CPolinomio
{
    friend ostream& operator<<(ostream&, CPolinomio&);

private:
    vector<CTermino *> termino; // matriz inicialmente vacía
public:
    CPolinomio(); // constructor
    CPolinomio(const CPolinomio&); // constructor copia
    ~CPolinomio(); // destructor
    CPolinomio& operator=(const CPolinomio&); // operador =
    size_t ObtenerNroTerminos() const;
    void AsignarTermino(CTermino);
    CPolinomio operator+(CPolinomio&);
    double operator()(double = 1, double = 1);
    operator double();
};
////////////////////////////////////

#endif // _CPOLINOMIO_H_

```

Como se puede observar, esta clase es igual que la diseñada en el capítulo anterior, excepto que ahora los elementos del vector son de tipo *CTermino **; esto es, se trata de una matriz de punteros a objetos *CTermino*. Esto lo hacemos así por dos razones: para que vea la diferencia que hay entre trabajar con una matriz de punteros a objetos y una matriz de objetos y para facilitarle la resolución de uno de los ejercicios que se proponen a continuación.

El constructor *CPolinomio* crea un objeto polinomio inicialmente con capacidad para diez términos, pero con un número inicial de términos igual a cero.

```

CPolinomio::CPolinomio()
{
    termino.reserve(10); // reservar memoria para 10 términos
}

```

El destructor libera la memoria asignada para cada objeto *CTermino*. La matriz de punteros será liberada por el destructor de *vector<...>*.

```

CPolinomio::~~CPolinomio()
{
    // Liberar la memoria ocupada por los términos del polinomio
}

```

```

    for (unsigned int i = 0; i < ObtenerNroTerminos(); i++)
        delete termino[i];
}

```

El constructor copia construye un nuevo objeto *CPolinomio*, idéntico a uno existente. Este método es requerido, por ejemplo, por una operación como:

```
PolinomioR = PolinomioA + PolinomioB;
```

Esta operación también requiere del operador de asignación que a continuación describimos. Obsérvese que este método primero inicia el polinomio destino con cero elementos y después añade, uno a uno, los términos del polinomio origen. Los objetos *CTermino* añadidos son un duplicado de los existentes; esto tiene que hacerse así porque si hiciéramos:

```
termino[i] = pol.termino[i]
```

los dos polinomios, origen y destino, harían referencia a los mismos términos, con lo cual, las modificaciones realizadas en uno de ellos repercutirían también de la misma forma en el otro, y cuando se eliminara uno de los polinomios, el otro se quedaría sin términos.

```

CPolinomio& CPolinomio::operator=(const CPolinomio& pol)
{
    // Iniciar a cero el polinomio destino.
    if (ObtenerNroTerminos())
    {
        for (unsigned int i = 0; i < ObtenerNroTerminos(); i++)
            delete termino[i];
        termino.clear();
    }
    // Copiar el polinomio origen en el nuevo destino
    for (unsigned int i = 0; i < pol.ObtenerNroTerminos(); i++)
        termino.push_back(new CTermino(*(pol.termino[i])));

    return *this;
}

```

Una explicación análoga daríamos para el constructor copia, excepto en que no hay un polinomio destino; precisamente es lo que queremos crear. Por eso, este método se limita a invocar al operador de asignación:

```

CPolinomio::CPolinomio(const CPolinomio& pol)
{
    *this = pol; // invoca al operador de asignación
}

```

El método *ObtenerNroTerminos* devuelve el número de términos del polinomio, o lo que es lo mismo, de la matriz *termino*.

```
size_t CPolinomio::ObtenerNroTerminos() const
{
    return termino.size();
}
```

Para añadir un nuevo término en el polinomio escribiremos el método *AsignarTermino*, que permite insertar el término pasado como argumento, en orden ascendente del exponente de x ; y a exponentes iguales de x , en orden ascendente de y . Este método primeramente verifica si el coeficiente del término a insertar es 0, en cuyo caso finaliza sin realizar ninguna inserción. Si el coeficiente es distinto de 0, verifica si el término en xy a insertar ya existe, en cuyo caso simplemente suma al coeficiente existente el del término pasado como argumento; si el resultado de esta suma es 0, invoca además al método **vector<...>::erase** para quitar ese término. Si el término no existe, entonces lo inserta en el lugar adecuado invocando al método **vector<...>::insert**.

Recuerde que tanto **insert** como **push_back** hacen una copia de su argumento en el vector destino (si el argumento es un puntero, copian el puntero y si fuera un objeto, copiarían el objeto).

```
void CPolinomio::AsignarTermino(CTermino t)
{
    // Asigna un término al polinomio colocándolo en orden ascendente
    // de los exponentes.
    if (t.ObtenerCoeficiente() == 0) return;

    float c, coef = t.ObtenerCoeficiente();
    int expx = t.ObtenerExponenteDeX();
    int expy = t.ObtenerExponenteDeY();

    // Insertar un nuevo término.
    int i = ObtenerNroTerminos() - 1;
    while (i >= 0 && expx < termino[i]->ObtenerExponenteDeX())
        i--;

    while (i >= 0 && expx == termino[i]->ObtenerExponenteDeX()
           && expy < termino[i]->ObtenerExponenteDeY())
        i--;

    if (i >= 0 && expx == termino[i]->ObtenerExponenteDeX()
        && expy == termino[i]->ObtenerExponenteDeY())
    {
        c = coef + termino[i]->ObtenerCoeficiente();
        // Término existente. Sumar los coeficientes.
        if (c)
```

```

        termino[i]->AsignarCoeficiente(c);
    else
        termino.erase(termino.begin()+i);
    }
    else
        // Insertar un nuevo término.
        termino.insert(termino.begin()+(i+1), new CTermino(t));
}

```

El siguiente método permite sumar dos polinomios. La idea básica es construir un tercer polinomio que contenga los términos de los otros dos, pero sumando los coeficientes de los términos que se repitan en ambos y desechando los términos cuyo coeficiente resultante sea 0. Los términos en el polinomio resultante también quedarán ordenados ascendentemente, por el mismo criterio que se expuso anteriormente. Un ejemplo de cómo invocar a este método puede ser el siguiente:

```
PolR = PolA + PolB;
```

El proceso de sumar consiste en:

- a) Partiendo de los polinomios *polA* y *polB* que se quieren sumar, obtener un término de cada uno de ellos.
- b) Comparar los dos términos (uno de cada polinomio) según el criterio explicado cuando se expuso el método *AsignarTermino*; si se trata del mismo término, sumar los coeficientes y almacenar ese término en *PolR*, sólo si el coeficiente es distinto de 0; si los términos son diferentes, almacenar en *polR* el que esté antes según el orden establecido.
- c) Obtener el siguiente término del polinomio al que pertenecía el término almacenado en *polR* y volver al punto b).
- d) Cuando no queden más elementos en uno de los dos polinomios de partida, se copian directamente en *polR* todos los elementos que queden en el otro polinomio.

```

CPolinomio CPolinomio::operator+(CPolinomio& polB)
{
    unsigned int ipa = 0, ipb = 0;
    int na = ObtenerNroTerminos(), nb = polB.ObtenerNroTerminos();
    float coefA, coefB;
    int expxA, expyA, expxB, expyB;
    CPolinomio polR;

    // Sumar polA con polB

```

```

while (ipa < na && ipb < nb)
{
    coefA = termino[ipa]->ObtenerCoeficiente();
    expxA = termino[ipa]->ObtenerExponenteDeX();
    expyA = termino[ipa]->ObtenerExponenteDeY();
    coefB = polB.termino[ipb]->ObtenerCoeficiente();
    expxB = polB.termino[ipb]->ObtenerExponenteDeX();
    expyB = polB.termino[ipb]->ObtenerExponenteDeY();
    if (expxA == expxB && expyA == expyB)
    {
        if (coefA + coefB != 0)
            polR.termino.push_back(new CTermino(coefA + coefB,
                                                expxA, expyA));

        ipa++, ipb++;
    }
    else if (expxA < expxB || (expxA == expxB && expyA < expyB))
    {
        polR.termino.push_back(new CTermino(coefA, expxA, expyA));
        ipa++;
    }
    else
    {
        polR.termino.push_back(new CTermino(coefB, expxB, expyB));
        ipb++;
    }
}
// Términos restantes de polA o de polB
while (ipa < na)
{
    coefA = termino[ipa]->ObtenerCoeficiente();
    expxA = termino[ipa]->ObtenerExponenteDeX();
    expyA = termino[ipa]->ObtenerExponenteDeY();
    polR.termino.push_back(new CTermino(coefA, expxA, expyA));
    ipa++;
}
while (ipb < nb)
{
    coefB = polB.termino[ipb]->ObtenerCoeficiente();
    expxB = polB.termino[ipb]->ObtenerExponenteDeX();
    expyB = polB.termino[ipb]->ObtenerExponenteDeY();
    polR.termino.push_back(new CTermino(coefB, expxB, expyB));
    ipb++;
}

return polR;
}

```

El siguiente método sobrecarga el operador de inserción para visualizar todos los términos del polinomio pasado como argumento.

```
ostream& operator<<(ostream& os, CPolinomio& polX)
{
    int i = polX.ObtenerNroTerminos();
    while (i--)
        polX.termino[i]->VisualizarTermino();
    return os;
}
```

Este otro método sobrecarga el operador función. Dicho método tiene dos parámetros que se corresponden con los valores de las variables x e y (por omisión, sus valores son 1) y devuelve como resultado el valor del polinomio.

```
double CPolinomio::operator()(double x, double y)
{
    double v = 0;
    for (unsigned int i = 0; i < ObtenerNroTerminos(); i++)
        v += termino[i]->ObtenerCoeficiente() *
            pow(x, termino[i]->ObtenerExponenteDeX()) *
            pow(y, termino[i]->ObtenerExponenteDeY());
    return v;
}
```

El siguiente método es un operador de conversión. Se invoca automáticamente siempre que una operación requiera convertir un objeto *CPolinomio* en un valor **double**. El valor **double** calculado es el valor del polinomio para $x=1$ e $y=1$.

```
CPolinomio::operator double()
{
    return (*this)(); // invoca al operador ()
}
```

El siguiente programa, utilizando las clases *CTermino* y *CPolinomio*, lee dos polinomios, visualiza estos polinomios así como el polinomio suma de ambos y también visualiza el valor del polinomio suma para $x=5$ e $y=1$ y para $x=1$ e $y=1$.

```
// polinomios.cpp - Trabajando con polinomios
//
#include <iostream>
#include "polinomio.h"
using namespace std;
bool IntroducirTermino(CTermino&);

int main() // función principal
{
    CPolinomio PolinomioR, PolinomioB, PolinomioA;
    CTermino tx;
    bool r;
    cout << "Términos del polinomio A\n"
```



```

        << "(para finalizar introduzca 0 para el\n"
        << "coeficiente y para los exponentes):\n\n";
r = IntroducirTermino(tx);
while (r)
{
    PolinomioA.AsignarTermino(tx); // duplica tx
    r = IntroducirTermino(tx);
}
cout << "Términos del polinomio B\n"
    << "(para finalizar introduzca 0 para el\n"
    << "coeficiente y para los exponentes):\n\n";
r = IntroducirTermino(tx);
while (r)
{
    PolinomioB.AsignarTermino(tx);
    r = IntroducirTermino(tx);
}

// Operador + y operador de asignación por omisión
PolinomioR = PolinomioA + PolinomioB;
// Constructor copia
CPolinomio Polinomio = PolinomioR;
cout << "\nPolinomio A: "; cout << PolinomioA;
cout << "\nPolinomio B: "; cout << PolinomioB;
cout << "\nPolinomio R: "; cout << Polinomio;
cout << endl;
cout << "valor del polinomio para x = 5, y = 1: "
    << PolinomioR(5, 1) << endl;
double v = PolinomioR; // valor del polinomio para x = 1 e y = 1
cout << "valor del polinomio para x = y = 1: " << v << endl;
}

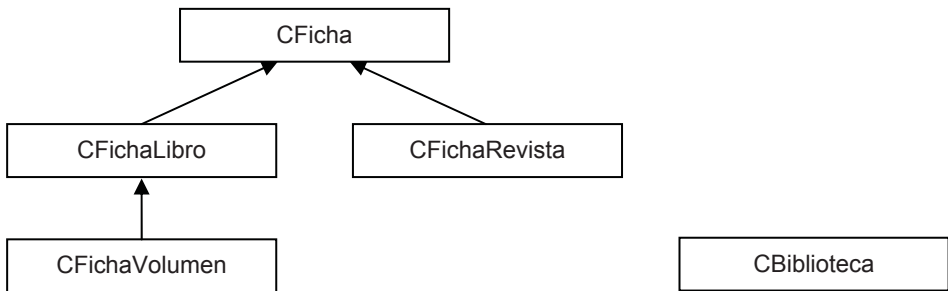
bool IntroducirTermino(CTermino& t)
{
    float coef;
    int expx, expy;
    cout << "Introduce coeficiente: "; cin >> coef;
    cout << "Introduce exponente de X: "; cin >> expx;
    cout << "Introduce exponente de Y: "; cin >> expy;
    cout << endl;
    if (!coef && !expx && !expy) return false;
    t = CTermino(coef, expx, expy);
    return true;
}

```

EJERCICIOS PROPUESTOS

1. El programa que se propone a continuación definirá una clase llamada *CFicha*, diseñada para manipular objetos de una biblioteca, como libros, revistas, obras de

varios volúmenes, DVD, etc. Todos estos objetos pueden tener en común un número que los identifique y un título. Según esto, la clase *CFicha* estará formada por los datos miembro *referencia* y *título* y por los métodos necesarios para manipularlos. La idea, como muestra la figura siguiente, es disponer de una clase para definir otras que amplíen su funcionalidad; por eso la definiremos abstracta.



Según lo expuesto, la clase *CFicha* podría ser así:

```

class CFicha
{
protected:
    std::string referencia;
    std::string titulo;
public:
    // Constructores
    CFicha(std::string = "", std::string = "");
    // Destructor
    virtual ~CFicha() {};
    // Otras funciones
    void AsignarReferencia(std::string);
    std::string ObtenerReferencia() const;
    void AsignarTitulo(std::string);
    std::string ObtenerTitulo() const;
    virtual CFicha *Clonar() = 0;
};

```

Pensemos ahora en un objeto particular; por ejemplo, un libro. Un libro tiene las características definidas por *CFicha*, y además algunas otras; por ejemplo, *autor* y *editorial*. Según esto, escribir una clase *CFichaLibro* derivada de *CFicha* que aporte estos nuevos datos y la funcionalidad necesaria para manipularlos.

Puesto que hay algunas obras compuestas por varios volúmenes (libros), podemos definir para este tipo de objetos una clase *CFichaVolumen*, derivada de la clase *CFichaLibro*, que aporte el *número de volumen* y la funcionalidad necesaria para manipularlo.

Supongamos también que tenemos revistas científicas y que para este tipo de objetos necesitamos almacenar, además de los datos referencia y título, el *número* de la revista y el *año* en que se publicó. Para ello definiremos una clase *CFichaRevista*, derivada de la clase *CFicha*, que aporte estos dos últimos datos y la funcionalidad necesaria para manipularlos.

Una vez construida la jerarquía de clases que se acaba de describir, escribir otra clase *CBiblioteca* que permita manipular objetos de esa jerarquía de clases. Para ello, la estructura de datos que represente la biblioteca tiene que ser capaz de almacenar objetos *CFichaLibro*, *CFichaVolumen* y *CFichaRevista*. Sabiendo que cualquier puntero a un objeto de una clase derivada puede convertirse implícitamente en un puntero a un objeto de su clase base, la estructura idónea será un **vector** de tipo *CFicha* *. Según esto, la clase *CBiblioteca* (que no pertenece a la jerarquía) tendrá un miembro que será un vector:

```
class CBiblioteca
{
    std::vector<CFicha *> ficha;
    // ...
};
```

Para finalizar, queda escribir un programa que, utilizando la clase *CBiblioteca* y la jerarquía de clases que tiene por raíz *CFicha*, construya la biblioteca objetivo del ejemplo propuesto. Este programa presentará un menú con, al menos, las opciones mostradas a continuación:

1. Añadir ficha
2. Buscar ficha
3. Buscar siguiente
4. Eliminar ficha
5. Listado de la biblioteca
6. Copia de seguridad de la biblioteca
7. Restaurar copia de seguridad
8. Salir

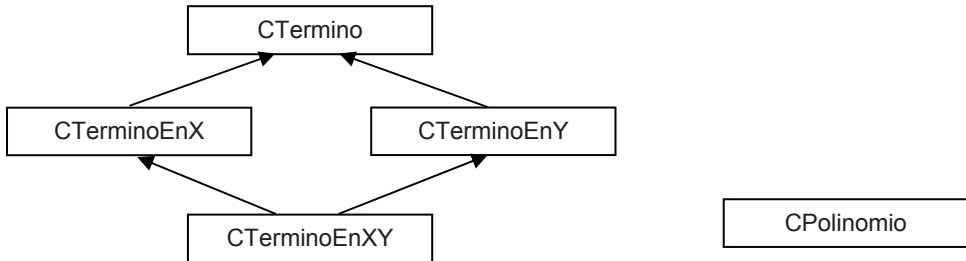
Las opciones 1 a 5 se refieren a un objeto libro, volumen o revista y las opciones 6 y 8 se refieren a un objeto biblioteca.

2. Se quiere escribir un programa para manipular ecuaciones algebraicas o polinómicas dependientes de las variables x y y . Por ejemplo:

$$2x^3y - xy^3 + 8.25 \text{ más } 5x^5y - 2x^3y + 7x^2 - 3 \text{ igual a } 5x^5y + 7x^2 - xy^3 + 5.25$$

Cada término del polinomio será representado por una clase *CTerminoEnX*, *CTerminoEnY* o *CTerminoEnXY* y cada polinomio por una clase *CPolinomio*. Las clases *CTerminoEnX* y *CTerminoEnY* se derivarán de una clase abstracta *CTermi-*

no y la clase *CTerminoEnXY* se derivará de las clases *CTerminoEnX* y *CTerminoEnY*.



Tenga en cuenta que ahora un objeto *CPolinomio* puede contener objetos de las clases *CTerminoEnX*, *CTerminoEnY* o *CTerminoEnXY*. No obstante, comprobará que si no trabaja con términos del mismo tipo, las operaciones con polinomios, como la suma, se complican excesivamente.

```

class CTermino // clase abstracta
{
private:
    double coeficiente;
public:
    CTermino(double k = 1) : coeficiente(k) {}
    virtual ~CTermino() {}
    float ObtenerCoeficiente() const;
    void AsignarCoeficiente(double coef = 1);
    virtual CTermino *clonar() = 0;
    // ...
};
  
```

Según la declaración de la clase *CTermino*, la clase *CTerminoEnX* tendrá sólo un atributo, *ExponenteDeX*, la clase *CTerminoEnY* tendrá también sólo un atributo, *ExponenteDeY*, y la clase *CTerminoEnXY* no necesita declarar atributos, puesto que hereda los anteriores.

De acuerdo con el enunciado y apoyándose en el ejercicio anteriormente resuelto, construya las clases a las que hemos hecho referencia para que soporten al menos la misma funcionalidad que vio allí y realice un programa similar al anterior, *polinomios.cpp*, para probar su funcionalidad.

PLANTILLAS

Hasta ahora, una función o una clase ha sido diseñada para trabajar con un tipo específico de datos. Los *tipos genéricos* o *tipos parametrizados*, también llamados *patrones* o *plantillas*, permiten construir una familia de funciones o de clases relacionadas, diferenciándose entre sí en el tipo de los datos que manipulan; esto es, son un mecanismo C++ que permite que un tipo pueda ser utilizado como parámetro en la definición de una función o de una clase. En realidad, este mecanismo tiene poco interés para el diseñador de la función o de la clase, pero tiene verdadera importancia para el usuario de esa función o clase, ya que le permitirá elegir el tipo de datos que necesite en cada momento.

Para ilustrar la definición que acabamos de exponer, vamos a intentar establecer una analogía utilizando el concepto de “tarta”. Una “tarta de <tipo>” podría ser una plantilla capaz de generar distintas clases de tartas: tarta de chocolate, tarta de manzana, tarta de moras, tarta de queso, etc. En realidad, “tarta” responde a un concepto genérico y las distintas particularizaciones forman una familia de productos relacionados. La idea que se obtiene de este ejemplo es que una plantilla, por medio de unos parámetros, permitirá generar distintas definiciones de clases o de funciones.

Según lo expuesto, resulta evidente que las *plantillas* simplifican la implementación de clases que definen contenedores, puesto que el tipo de los objetos contenidos es un argumento en la definición de la clase; por ejemplo, las listas dinámicas y las matrices son ejemplos de contenedores. También permiten definir funciones genéricas para trabajar con un amplio número de tipos. Piense, por ejemplo, en una función *ordenar* que le brinde la posibilidad de elegir el tipo de los elementos que quiere ordenar.

Una programación que utiliza tipos como parámetros recibe el nombre de *programación genérica*. La idea de este capítulo es introducirle en el diseño, im-

plementación y utilización de plantillas. La biblioteca estándar de C++, según hemos podido comprobar en los capítulos anteriores, presenta muchas de sus abstracciones como plantillas; por ejemplo, **basic_string**, **vector** y **map**.

DEFINICIÓN DE UNA PLANTILLA

La definición de una plantilla, de clase o de función, se hace según la siguiente sintaxis:

template<*lista de parámetros*> *declaración*

La palabra reservada de C++ **template** indica que la *declaración* especificada a continuación es una plantilla. La lista de parámetros de la plantilla se especifica entre $\langle \rangle$ y consta de una serie de identificadores calificados separados por comas. Y *declaración* hace referencia a la declaración o a la definición de la plantilla. Por ejemplo, el código siguiente declara una plantilla *fx* de función y otra *X* de clase:

```
template<class T, T p2, int p3> T fx( T x );
template<class P1, template<class T> class P2,
        class P3 = vector<P1> > class X;    // espacio entre > y >
```

Estas declaraciones tendrán sus correspondientes definiciones en alguna otra parte del código (definiciones análogas a las definiciones de función y clase que ya conocemos, pero basadas en su lista de parámetros) que genéricamente denominamos plantillas. El compilador basándose en estas plantillas podrá generar implícitamente funciones y clases particularizadas para unos valores específicos de sus parámetros, pasados como argumentos.

Los parámetros de una plantilla pueden ser:

- *Identificadores de tipo*, por ejemplo *T*. Estos parámetros van precedidos por la palabra reservada **class** o **typename** especificando que *id* es un tipo:

```
class id
class id = id-de-tipo
typename id
typename id = id-de-tipo
```

- *Plantillas de clases*. Un parámetro puede ser una plantilla de clase, pero no una plantilla de función:

```
template<lista de parámetros> class id
template<lista de parámetros> class id = id-plantilla-clase
```

- Parámetros de algún *otro tipo*: primitivo, derivado, definido por el usuario o de plantilla:

```
tipo id
tipo id = valor
```

Un parámetro de plantilla que no sea un tipo es una constante dentro de la plantilla, por lo tanto, no se puede modificar.

Cuando se genera una clase o una función a partir de una plantilla, los argumentos pasados que no sean tipos pueden ser: una expresión constante, la dirección de un objeto (de la forma *&objeto*) o de una función (de la forma *f*, siendo *f* el nombre de la función) o de un miembro no sobrecargado (de la forma *&C::miembro*). Un literal de cadena de caracteres no es válido como argumento. Por ejemplo:

```
template<class T> class MiVector { ... };
X<int, MiVector> c; // objeto c de la clase X<int, MiVector>
int a, b = 10;
a = fx<int, 3, 5>(b); // invoca a la función fx<int, 3, 5>
```

Cuando se compila una plantilla, su definición se verifica en cuanto a errores de sintaxis se refiere y en cuanto a otros errores que puedan ser detectados, como, por ejemplo, un parámetro iniciado con un valor no válido. En cambio, los errores relacionados con el uso de los parámetros de plantilla no pueden detectarse hasta que la plantilla se utilice.

La palabra reservada **typename** se utiliza para declarar que un identificador es un tipo, por lo tanto puede sustituir sin ningún problema a **class**. También se utiliza en los casos en los que el compilador no puede adivinar si un identificador es un tipo. Por ejemplo, es necesario utilizar esta palabra reservada cada vez que un identificador de tipo depende de un parámetro de plantilla:

```
template<class T> void fx(vector<T>& v)
{
    typename vector<T>::iterator it = v.begin();
    // ...
}
```

En este ejemplo, *iterator* es un identificador dependiente del parámetro *T*, razón por la cual se requiere **typename**, ya que en este caso el compilador no tiene capacidad para determinar que *iterator* es un nombre de un tipo.

Finalmente, obsérvese en los ejemplos iniciales que los parámetros de las plantillas de función no pueden tener valores por omisión, y sí los pueden tener las plantillas de clase.

FUNCIONES GENÉRICAS

Una función genérica define una familia de funciones en base a una plantilla de función. Por ejemplo, consideremos la familia de funciones *menor(a, b)* que retorna el valor más pequeño de sus dos argumentos. Una versión explícita de esta familia puede ser la siguiente:

```
int menor(int a, int b)
{
    return (a < b) ? a : b;
}
```

Esta función permite comparar dos enteros, pero no funcionaría correctamente para un caso como el siguiente, porque un objeto *CFecha* no se puede convertir en un **int**:

```
CFecha f1, f2;
// ...
cout << menor(f1, f2) << endl;
```

Evidentemente, estamos suponiendo que los datos utilizados son de un tipo capaz de ser ordenado. Por lo tanto, para este caso, la clase *CFecha* debe incluir un operador *<* (una fecha es menor que otra si es anterior). Partiendo de este supuesto, la solución al problema planteado es escribir otra versión explícita de la función como la siguiente:

```
CFecha menor(CFecha a, CFecha b)
{
    return (a < b) ? a : b;
}
```

Pero el problema volverá a surgir en cuanto trabajemos con otro tipo de objetos diferente a **int** o a *CFecha*. Por lo tanto, la mejor solución sería implementar una *función genérica*. Esto implica escribir una plantilla de función en la que el tipo de los objetos a comparar sea un parámetro. Por ejemplo, la declaración de la función *menor* puede escribirse así:

```
template<class T> T menor(T a, T b);
```

Y su definición, así:

```
template<class T> T menor(T a, T b)
{
    return (a < b) ? a : b;
}
```


El hecho de especificar la declaración y la definición de una plantilla es porque, análogamente a como hacíamos con las funciones, las plantillas de función pueden incluirse en una unidad de traducción de estas dos formas:

1. Primero se definen y después se utilizan.
2. Primero se declaran, después se utilizan y finalmente se definen.

El programa que se muestra a continuación incluye una plantilla de función denominada *menor* de la forma 1. Este programa utiliza la clase *CFecha* diseñada en capítulos anteriores, a la que se ha añadido el operador `<`.

```
// test.cpp - Plantilla de función
#include "fecha.h"
#include <iostream>
using namespace std;
```

```
template<class T> T menor(T a, T b)
{
    return (a < b) ? a : b;
}
```

```
int main()
{
    int m = 10, n = 27;
    CFecha f1(20), f2(15);
```

```
    int r = menor(m, n);
    CFecha f = menor(f1, f2);
    cout << r << endl;
    cout << f.obtenerDia() << endl;
}
```

Una función genérica no es una definición de función, sino una plantilla desde la que el compilador puede generar funciones implícitas. Por lo tanto, durante la ejecución no existe nada parecido a funciones genéricas, sólo existen funciones concretas. Por ejemplo, cuando el compilador resuelve la llamada *menor*(*m*, *n*) del ejemplo anterior, generará de forma automática la siguiente función, siempre y cuando no esté ya generada:

```
int menor(int a, int b)
{
    return (a < b) ? a : b;
}
```

Esta función se genera a partir de la plantilla *menor* para *T* igual a **int**. ¿Por qué ésta y no otra? Porque *m* y *n* son de tipo **int**. Análogamente, la llamada *menor*(*f1*, *f2*) se resuelve así:

```
CFecha menor(CFecha a, CFecha b)
{
    return (a < b) ? a : b;
}
```

Hay que tener presente que no todos los tipos tienen un operador `<`, lo que significa que la plantilla puede utilizarse sólo con tipos que definan este operador. Esto es, el criterio que define qué objeto *a* o *b* es menor no está definido en *menor*, sino en la propia clase de los objetos. Sólo bajo esta premisa podremos escribir algoritmos genéricos.

Se puede observar que los valores para los parámetros de la plantilla se deducen de los argumentos pasados a la función, lo que también determina la versión de la plantilla que se utiliza. Esto obliga a que todos los parámetros de la plantilla (los especificados entre `<>`) deben estar representados en los parámetros formales de la función (los especificados entre `()`). De no ser así, no se podrían deducir aquéllos que no aparecen, ya que no hay ningún argumento que los identifique. Por ejemplo, la plantilla *asignarMem* que se muestra a continuación tiene un parámetro *T* que no se utiliza en la lista de parámetros formales de la función, por lo que no es posible deducir de la llamada su valor:

```
template<class T> T *asignarMem(int tam)
{
    T *p = 0;
    try
    {
        p = new T[tam];
        fill(p, p + tam, 0);
    }
    catch (bad_alloc)
    {
        cout << "Insuficiente memoria\n";
        exit(-1);
    }
    return p;
}

int main()
{
    int tm = 10;
    double *pd;
    pd = asignarMem(tm); // error
    // ...
    delete pd;
}
```

Como podremos comprobar más adelante, lo anterior no es aplicable a los métodos de clases genéricas, ya que los valores de los parámetros de plantilla son proporcionados cuando se concreta la plantilla en una clase por primera vez.

Cuando no se pueda deducir el valor de un parámetro de la plantilla de los argumentos pasados a la función, éste debe especificarse de manera explícita entre `<>` después del nombre de la plantilla. Por ejemplo:

```
int main()
{
    int tm = 10;
    double *pd;
    pd = asignarMem<double>(tm);
    // ...
    delete pd;
}
```

La expresión `asignarMem<double>` genera la versión de `asignarMem` para `T` igual a `double`. Otra solución podría ser escribir la plantilla de función así:

```
template<class T> void asignarMem(T **p, int tam)
{
    try
    {
        *p = new T[tam];
        fill(*p, *p + tam, 0);
    }
    catch (bad_alloc)
    {
        cout << "Insuficiente memoria\n";
        exit(-1);
    }
}

int main()
{
    int tm = 10;
    double *pd;
    asignarMem(&pd, tm);
    // ...
    delete pd;
}
```

Evidentemente, el número de parámetros de tipo en una plantilla puede ser cualquiera, independientemente de que en los ejemplos realizados hayamos utilizado sólo uno.

Especialización de plantillas de función

Una versión de una plantilla para un parámetro de plantilla concreto se denomina *especialización*. Veamos un ejemplo que aclare este concepto:

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

template<class T> T menor(T a, T b) // plantilla general
{
    return (a < b) ? a : b;
}

int main()
{
    char *cad1 = "hola";
    char *cad2 = "adiós";
    char *cad3 = menor(cad1, cad2); // cad3 = "hola"
}
```

¿Cuál sería el resultado después de ejecutar el código anterior? Lógicamente, el resultado no es el esperado, porque *menor* comparará las direcciones de las cadenas y no las cadenas. Evidentemente esto lo podríamos solucionar añadiendo una función particularizada para este tipo de datos, como la siguiente:

```
char *menor(char *a, char *b)
{
    return (strcmp(a, b) < 0) ? a : b;
}
```

Ahora el resultado sería correcto, pero qué pasaría si un usuario escribe una llamada como la siguiente, en vez de la llamada anterior:

```
int main()
{
    char *cad1 = "hola";
    char *cad2 = "adiós";
    char *cad3 = menor<char *>(cad1, cad2);
}
```

Pues, lo que sucede es que el resultado vuelve a ser incorrecto, porque esta llamada no utiliza la función *menor* añadida anteriormente, sino que genera a partir de la plantilla *menor* otra función para *T* igual a **char *** (obsérvese que ésta última utiliza **<** y no **strcmp**). La solución está en ofrecer una definición alternativa de la plantilla *menor* particularizada para *T* igual a **char ***. Dicha definición se denomina *especialización* y sería así:

```
template<> char *menor<char *>(char *a, char *b)
{
    return (strcmp(a, b) < 0) ? a : b;
}
```

El prefijo `template<>` indica que la definición es una *especialización explícita* de la plantilla. Los valores de los parámetros para los que se utilizará la plantilla se especifican de forma explícita a continuación del nombre entre `<>`; es decir, `<char *>` indica que esta plantilla se va a utilizar para generar una función *menor* cuando el valor pasado en la llamada para el parámetro *T* sea `char *`. Una especialización es una plantilla (no una función generada por el compilador a partir de una plantilla) y tiene que definirse después de la plantilla general.

Como el valor para el parámetro de plantilla puede deducirse a partir de la lista de parámetros de la función, la lista entre ángulos después del nombre de la plantilla es redundante y puede omitirse:

```
template<> char *menor(char *a, char *b)
{
    return (strcmp(a, b) < 0) ? a : b;
}
```

Entre lo general y lo específico siempre hay un punto intermedio (lo que el estándar C++ denomina especialización parcial). Por ejemplo, para definir una plantilla que sólo compare vectores de elementos de tipo *T*, fijamos de forma parcial los parámetros de la plantilla:

```
template<class T> vector<T> menor(vector<T> a, vector<T> b)
{
    return (a < b) ? a : b;
}
```

La sintaxis para invocar a esta especialización sería:

```
menor<T>(vector<T>, vector<T>);
```

En realidad, la plantilla anterior es una sobrecarga (como veremos a continuación) y no una especialización parcial, aunque actúe como tal. Con plantillas de función sólo cabe hablar de *especializaciones explícitas* como `menor(char *, char *)` o de sobrecargas de plantillas como `menor(T *, T *)`.

Si la comparación fuera sólo de vectores de un tipo específico, la especialización sería explícita. Por ejemplo:

```
template<> vector<double> menor(vector<double> a, vector<double> b)
{
```

```

    return (a < b) ? a : b;
}

int main()
{
    vector<double> v1(10);
    vector<double> v2(8);
1. menor<double>(v1, v2); // utiliza la plantilla explícita
2. menor<vector<double> >(v1, v2); // utiliza la plantilla general
3. menor(v1, v2); // utiliza la plantilla explícita

    vector<int> v3(10);
    vector<int> v4(8);
4. menor(v3, v4); // utiliza la sobrecarga de la plantilla
}

```

En el ejemplo anterior, la función **main** llama cuatro veces a *menor*. Los prototipos de las llamadas realizadas, en función del argumento explícito especificado y/o de los valores deducidos de los argumentos pasados a la función, son:

1. *menor<double>(vector<double>, vector<double>)* que utiliza la plantilla explícita para vectores de tipo **double**.
2. *menor<vector<double> >(vector<double>, vector<double>)* que utiliza la plantilla general para $T = \text{vector}<\text{double}>$.
3. *menor<double>(vector<double>, vector<double>)* que utiliza la plantilla explícita para vectores de tipo **double**.
4. *menor<int>(vector<int>, vector<int>)* que utiliza la sobrecarga de la plantilla para $T = \text{int}$.

Sobrecarga de plantillas de función

Las plantillas de función se pueden sobrecargar de forma análoga a como sobrecargamos las funciones normales. Incluso pueden combinarse sobrecargas de plantillas y funciones con el mismo nombre. Por ejemplo:

```

template<class T> T menor(T a, T b)
{
    return (a < b) ? a : b;
}

template<class T> vector<T> menor(vector<T> a, vector<T> b)
{
    return (a < b) ? a : b;
}

```

```

double menor(double a, double b)
{
    return (a < b) ? a : b;
}

int main()
{
    vector<double> v1(10);
    vector<double> v2(8);

    menor(10, 27); // menor<int>(int, int)
    menor(v1, v2); // menor<double>(vector<double>, vector<double>)
    menor(26.2, 26.8); // menor(double, double)
}

```

En un caso como el que presenta el ejemplo anterior, ¿cómo resuelve el compilador a qué función tiene que invocar? Cuando utilizamos plantillas sobrecargadas, las reglas que se siguen son generalizaciones de las existentes para funciones normales, y que podemos resumir así:

1. En función de los argumentos especificados en las llamadas, se forma un conjunto de especializaciones de plantillas de función, eligiendo siempre entre las plantillas más especializadas, cuando hay varias. Por ejemplo, para la llamada *menor(v1, v2)* son candidatas las dos especializaciones siguientes:

```

menor<vector<double> >(vector<double> a, vector<double> b);
menor<double>(vector<double> a, vector<double> b);

```

Entre estas dos especializaciones es más especializada la segunda, porque cualquier llamada que coincida con *menor<T>(vector<T> a, vector<T> b)* también coincide con *menor<T>(T a, T b)*, pero no a la inversa.

2. Al conjunto de especializaciones anterior se añaden las funciones normales y se resuelve qué función tiene que llamarse como se hacía con las funciones ordinarias; esto es: **a)** correspondencia exacta, **b)** correspondencia utilizando promociones integrales (por ejemplo, de **char** a **int**, de **float** a **double**, etc.), **c)** correspondencia utilizando conversiones estándar (por ejemplo, de **int** a **double**, de puntero a derivada a puntero a base, etc.), **d)** correspondencia utilizando conversiones definidas por el usuario y **e)** correspondencia utilizando los puntos suspensivos (... => número variable de parámetros) en una declaración de función. Los puntos **b**, **c** y **d** no se aplican cuando el parámetro de una plantilla de función se ha determinado por deducción; en los demás casos, sí.
3. Entre una función y una especialización igualmente buenas, se toma la función. Por ejemplo, entre las dos llamadas siguientes es preferible la segunda:

```
menor<double>(double, double); // especialización
menor(double, double);       // función normal
```

4. Una vez aplicado el descarte que se hace en 3, si aún hay dos o más coincidencias que satisfagan la llamada, se produce un error de ambigüedad, y si no hay ninguna coincidencia, también se produce un error.

Las ambigüedades se pueden resolver de dos formas: realizando las conversiones explícitas necesarias sobre los argumentos pasados o añadiendo nuevas sobrecargas que satisfagan las llamadas.

ORGANIZACIÓN DEL CÓDIGO DE LAS PLANTILLAS

Para organizar el código fuente de las plantillas podemos seguir alguno de los métodos siguientes:

1. Fichero único.
2. Fichero de declaraciones y fichero de definiciones.
3. Fichero único combinación de otros.

Fichero único

Como norma general podemos decir que tanto las declaraciones como las definiciones que forman parte del código fuente que describe una plantilla, de clase o de función, tienen que estar presentes en cada unidad de traducción que utilice dicha plantilla. Esto es así porque es responsabilidad del compilador generar el código cuando sea necesario, razón por la que no se puede esperar a la fase de enlace, excepto si exportamos las plantillas (cláusula **export**), de lo cual hablaremos posteriormente (la estrategia seguida con las funciones genéricas es análoga a la seguida con las funciones **inline**). Lo que sí se hará durante la fase de enlace es filtrar el código generado en las distintas unidades de traducción para que sólo exista una definición de cada función o de cada clase. Veamos un ejemplo:

```
// plantillas.h - Definiciones de plantillas
#ifndef _PLANTILLAS_H_
#define _PLANTILLAS_H_

#include <vector>

template<class T> T menor(T a, T b)
{
    return (a < b) ? a : b;
}
```



```

template<class T>
std::vector<T> menor(std::vector<T> a, std::vector<T> b)
{
    return (a < b) ? a : b;
}

double menor(double a, double b)
{
    return (a < b) ? a : b;
}

#endif // _PLANTILLAS_H_

```

Observe que el fichero de cabecera *plantillas.h* incluye distintas formas de la plantilla *menor*, así como una función *menor* concreta. Ahora, cualquier unidad de traducción que requiera utilizar estas plantillas y/o funciones simplemente tendrá que incluir ese fichero de cabecera. Por ejemplo:

```

// test.cpp - Aplicación
#include <iostream>
#include <vector>
#include "plantillas.h"
using namespace std;

int main()
{
    vector<double> v1(10);
    vector<double> v2(8);

    cout << menor(10, 27) << '\n';
    vector<double> v3 = menor(v1, v2);
    cout << v3.size() << '\n';
    cout << menor(26.2, 26.8) << '\n';
}

```

Fichero de declaraciones y fichero de definiciones

La utilización de un fichero único presenta un problema y es que todo de lo que dependa el código generado (funciones o clases) también será añadido a cada unidad de traducción que utilice ese código, con lo que la cantidad de información que el compilador deberá procesar se multiplicará. Por el contrario, si las definiciones de las plantillas, análogamente a como se procede con las funciones normales, no se incluyeran en el código del usuario, ninguna de sus dependencias podrían afectar a ese código. Estamos hablando de incluir las declaraciones de las plantillas en un fichero de cabecera y las definiciones en un fichero *.cpp*. En este caso, el fichero *.cpp* se compilará por separado, igual que el resto de las unidades de traducción, y será responsabilidad del compilador encontrar las definiciones

cuando sean necesarias. Evidentemente, ahora el compilador, en vez de filtrar definiciones redundantes, deberá encontrar las definiciones requeridas. En este caso, la disposición de los ficheros será así:

```
// plantillas.h - Declaraciones de plantillas
#if !defined( _PLANTILLAS_H_ )
#define _PLANTILLAS_H_

#include <vector>

template<class T> T menor(T, T);
template<class T>
std::vector<T> menor(std::vector<T>, std::vector<T>);
double menor(double, double);

#endif // _PLANTILLAS_H_
```

```
// plantillas.cpp - Definiciones de plantillas
#include <vector>
using namespace std;

export template<class T> T menor(T a, T b)
{
    return (a < b) ? a : b;
}

export template<class T> vector<T> menor(vector<T> a, vector<T> b)
{
    return (a < b) ? a : b;
}

double menor(double a, double b)
{
    return (a < b) ? a : b;
}
```

Obsérvese que las plantillas son declaradas explícitamente exportables. Esto se hace añadiendo la palabra clave **export** a la definición o a la declaración.

```
// test.cpp - Aplicación
#include <iostream>
#include <vector>
#include "plantillas.h"
using namespace std;

int main()
{
    vector<double> v1(10);
    vector<double> v2(8);
```

```

cout << menor(10, 27) << '\n';
vector<double> v3 = menor(v1, v2);
cout << v3.size() << '\n';
cout << menor(26.2, 26.8) << '\n';
}

```

El fichero de cabecera sólo tiene las declaraciones de las plantillas (y cualquier otra declaración necesaria), el fichero *plantillas.cpp* contiene las definiciones de las plantillas (y cualquier otra definición necesaria), y cualquier otro fichero que forme parte de la aplicación y que requiera de esas definiciones sólo tendrá que incluir el fichero de cabecera. Evidentemente, el fichero *plantillas.cpp* formará parte del proyecto. En el ejemplo anterior, el proyecto está formado por los ficheros *plantillas.cpp* y *test.cpp*. Para compilar y enlazar estos ficheros, utilice la orden siguiente:

```
g++ test.cpp plantillas.cpp -o test.exe
```

Nota: en el momento de publicar este libro, los compiladores que había en el mercado no soportaban **export** (salvo raras excepciones, como el compilador de *Greg Comeau*). Esto demuestra que a pesar de que el estándar incluye esta característica, su implementación es compleja y presenta problemas. No obstante, la falta de **export** en un compilador sería lo que menos debiera preocuparle.

Fichero único combinación de otros

Con este modelo el autor trata de que el lector siga escribiendo las plantillas de clases, que estudiamos a continuación, análogamente a como escribe las clases: la declaración de la plantilla en un fichero de cabecera y su definición en un fichero *.cpp*. Se trata de seguir el modelo que nos proporciona la existencia de **export**, pero sin utilizar esta característica. Esto es, la disposición que proponemos de los ficheros es así:

```

// vector.h - Clase genérica Vector
#if !defined( _VECTOR_H_ )
#define _VECTOR_H_

template<class T> class Vector // declaración
{
    // ...
};

#include "vector.cpp" // definición

#endif // _VECTOR_H_

```

```
// test.cpp - Aplicación
#include "vector.h"

int main()
{
    // ...
}
```

El fichero de cabecera, *vector.h*, declara la plantilla (y cualquier otra declaración necesaria) e incluye al fichero *vector.cpp* que contiene las definiciones de la plantilla (y cualquier otra definición necesaria). Siguiendo el modelo al que ya estamos acostumbrados, hemos separado la declaración de la clase genérica de su definición, pero después, por tratarse de plantillas, ambas partes serán incluidas por el fichero de cabecera, en cualquier otro fichero que forme parte de la aplicación y que requiera de esas definiciones (obsérvese *test.cpp*). Evidentemente, en este caso, el fichero *vector.cpp* no formará parte del proyecto. Entonces, para compilar y enlazar la aplicación del ejemplo anterior, tendría que utilizar la orden siguiente:

```
g++ test.cpp -o test.exe
```

CLASES GENÉRICAS

Una *clase genérica* es una plantilla para definir un conjunto de clases que se diferenciarán entre sí en el tipo de los datos que manipulan. Un buen ejemplo son los contenedores (como las matrices), ya que independientemente del tipo de los datos que almacenan sus elementos, las operaciones básicas que permiten su manipulación son siempre las mismas: insertar, borrar, acceder a un elemento, etc. Por eso, la mejor forma de definir este tipo de estructuras de datos es utilizando *clases genéricas*, donde el tipo de los elementos sea un parámetro de la plantilla.

Para ilustrarlo, recordemos la clase *CVector* que escribimos en capítulos anteriores, con la intención de escribir una plantilla *Vector*:

```
class CVector
{
private:
    double *vector; // puntero al primer elemento de la matriz
    int nElementos; // número de elementos de la matriz
protected:
    double *asignarMem(int);
public:
    CVector(int ne = 10); // crea un CVector con ne elementos
    CVector(const CVector&); // crea un CVector desde otro
    ~CVector() { delete [] vector; } // destructor
    CVector& operator=(const CVector&);
```

```

    double& operator[](int i) { return vector[i]; }
    int longitud() const { return nElementos; }
};

```

Normalmente, es una buena idea escribir una plantilla de clase partiendo de una clase particular ya depurada. De esta forma, es más fácil solucionar los problemas de diseño y la mayor parte de los errores de código. Esto es justamente lo que hemos hecho. Cuando ya tenemos depurada la clase, pasamos a escribir la plantilla. Así, los errores que puedan surgir de este proceso podrán ser tratados sin sufrir la distracción derivada de los errores convencionales.

¿Cuántos parámetros tendrá nuestra plantilla *Vector*? Uno: el tipo de los elementos de la matriz que encapsula un objeto *CVector*; entonces, este tipo que en la clase *CVector* era **double** pasará a ser *T* en la plantilla *Vector*:

```

// vector.h - Plantilla de clase Vector
#if !defined( _VECTOR_H_ )
#define _VECTOR_H_

template<class T> class Vector // declaración
{
private:
    T *vector; // puntero al primer elemento de la matriz
    int nElementos; // número de elementos de la matriz
protected:
    T *asignarMem(int);
public:
    Vector(int ne = 10); // crea un Vector con ne elementos
    Vector(const Vector&); // crea un Vector desde otro
    ~Vector() { delete [] vector; vector = 0; } // destructor
    Vector& operator=(const Vector&);
    T& operator[](int i) { return vector[i]; }
    int longitud() const { return nElementos; }
};

#include "vector.cpp" // definición

#endif // _VECTOR_H_

```

El código anterior define la plantilla *Vector*, a través de la cual el compilador podrá generar una colección de clases *Vector<T>*. Por ejemplo, la clase *Vector<double>*, *Vector<complex<double>>*, etc. Obsérvese que al final de la declaración de la plantilla se incluyen las definiciones de los miembros de la misma.

Los miembros de una plantilla de clase se definen exactamente igual que los de cualquier otra clase, pero teniendo presente que éstos, a su vez, son plantillas parametrizadas con los mismos parámetros de la plantilla de clase; por lo tanto,

cuando se definen fuera de la declaración de la plantilla de clase, y sólo cuando se definen fuera, se deben definir exactamente igual que las plantillas de función.

Obsérvese la declaración de la plantilla *Vector*; incluye la definición de tres de sus miembros: *destructor*, *operador de indexación* y *longitud*. Los otros miembros sólo se han declarado y se definen fuera de la declaración de la plantilla:

```
// vector.cpp - Definición de la plantilla Vector
#include <iostream>
using namespace std;

// Constructores:
// Crear una matriz con ne elementos
template<class T> Vector<T>::Vector(int ne)
{
    if (ne < 1)
    {
        cerr << "Nº de elementos no válido: " << ne << "\n";
        return;
    }
    nElementos = ne;
    vector = asignarMem(nElementos);
}

// Constructor copia
template<class T> Vector<T>::Vector(const Vector& v) : vector(0)
{
    *this = v;
}

// Operador de asignación
template<class T> Vector<T>& Vector<T>::operator=(const Vector& v)
{
    nElementos = v.nElementos;           // número de elementos
    delete [] vector;                    // borrar la matriz actual
    vector = asignarMem(nElementos);     // crear una nueva matriz
    for (int i = 0; i < nElementos; i++)
        vector[i] = v.vector[i];        // copiar los valores
    return *this;                        // permitir asignaciones encadenadas
}

// Otros métodos
template<class T> T *Vector<T>::asignarMem(int nElems)
{
    try
    {
        T *p = new T[nElems];
        return p;
    }
}
```

```

catch(bad_alloc e)
{
    cout << "Insuficiente espacio de memoria\n";
    exit(-1);
}
}

```

Se puede observar que la pertenencia de un método a su clase queda resuelta por el nombre de la misma, que según vimos anteriormente es *Vector<T>*, más el operador de ámbito. Así mismo, dentro del ámbito de *Vector<T>*, la calificación *<T>* es redundante. Esto quiere decir que podríamos haber escrito el constructor también así:

```

template<class T> Vector<T>::Vector<T>(int ne)
{
    if (ne < 1)
    {
        cerr << "Nº de elementos no válido: " << ne << "\n";
        return;
    }
    nElementos = ne;
    vector = asignarMem(nElementos);
}

```

El nombre de una plantilla de clase es único en el ámbito donde se haya definido; dicho de otra forma, no existe la sobrecarga de plantillas de clase.

El siguiente ejemplo define un objeto *vector* de la clase *Vector<double>* que encapsula una matriz de cinco elementos de tipo **double**. Después asigna valores a cada uno de los elementos de la matriz y, finalmente, la visualiza.

```

// test.cpp - Aplicación
#include <iostream>
#include <iomanip>
#include "vector.h"
using namespace std;

template<class T> void visualizar(Vector<T>&);

int main()
{
    Vector<double> vector(5);

    for (int i = 0; i < vector.longitud(); i++)
        vector[i] = i+1;
    visualizar(vector);
}

```

```

template<class T> void visualizar(Vector<T>& v)
{
    int ne = v.longitud();
    for (int i = 0; i < ne; i++)
        cout << setw(7) << v[i];
    cout << "\n\n";
}

```

Cuando se compila la declaración `Vector<double> vector(5)` el compilador genera, a partir de la plantilla, la clase `Vector<double>` con todos sus miembros:

```

class Vector<double>
{
private:
    double *vector; // puntero al primer elemento de la matriz
    int nElementos; // número de elementos de la matriz
protected:
    double *asignarMem(int);
public:
    Vector(int ne = 10); // crea un Vector con ne elementos
    Vector(const Vector&); // crea un Vector desde otro
    ~Vector() { delete [] vector; vector = 0; } // destructor
    Vector& operator=(const Vector&);
    double& operator[](int i) { return vector[i]; }
    int longitud() const { return nElementos; }
};

```

En cuanto a los miembros definidos fuera de la declaración, deberían generarse sólo aquéllos que se utilizan, pero esto depende del compilador y, por lo general, éstos replican el código.

Si quisiéramos, por ejemplo, que el tipo *T* de los elementos de *vector* fuera, en lugar de **double**, una estructura con los miembros *nombre* y *teléfono*, procederíamos como se muestra a continuación:

```

// test.cpp - Aplicación
#include <iostream>
#include <string>
#include "vector.h"

using namespace std;

struct elemento
{
    string nombre;
    long telefono;
};

```



```

int main()
{
    Vector<elemento> vector(10); // crea una matriz de estructuras

    for (int i = 0; i < vector.longitud(); i++)
    {
        cout << "Nombre:    "; getline(cin, vector[i].nombre);
        cout << "Teléfono:  "; cin >> vector[i].telefono; cin.ignore();
    }

    for (int i = 0; i < vector.longitud(); i++)
    {
        cout << "Nombre:    " << vector[i].nombre << endl;
        cout << "Teléfono:  " << vector[i].telefono << endl;
    }
}

```

Declaración previa de una clase genérica

Una clase genérica o plantilla de clase puede tener una declaración adelantada para ser definida después, pero teniendo presente que esta definición debe estar antes de su utilización. Por ejemplo:

```
template<class T> class A; // declaración previa de A
```

```

template<class T> class B // definición de B
{
    friend class A<T>;
    // ...
};

```

```
template<class T> class A // definición de A
```

```

{
    // ...
};

int main()
{
    A<int> objA; // utilización de las plantillas
    B<int> objB;
}

```

Especialización de plantillas de clase

Por lo estudiado hasta ahora, sabemos que una plantilla es una definición única para utilizar con muchos tipos de datos diferentes. Por ejemplo, algunos usos de la plantilla *Vector* pueden ser los siguientes:

```

Vector<double> vector1;
Vector<double *> vector2;
Vector<CFecha> vector3;
Vector<CFecha *> vector4;
Vector<char> vector5;
Vector<string *> vector6;

```

El compilador, basándose en los argumentos pasados a la plantilla, genera la clase adecuada para el tipo de datos seleccionado por el usuario, replicando el código para las plantillas de función. Ahora bien, esto tiene sus ventajas y sus inconvenientes; ventajas, que es bueno para el rendimiento durante la ejecución; inconvenientes, que el código de la aplicación puede ser mucho más grande que lo esperado y que en ocasiones desearíamos utilizar no la implementación general, sino una implementación específica para un tipo de datos concreto; por ejemplo, las matrices de punteros podrían compartir una implementación específica.

Una versión de una plantilla para un parámetro de plantilla concreto se denomina *especialización*, y es una forma de implementar alternativas para diferentes usos de una interfaz común. Antes de cualquier especialización es preciso declarar la plantilla general. Veamos un ejemplo que aclare este concepto, partiendo de la plantilla general de *Vector*:

```

// vector.h - Plantilla general de clase Vector
#ifndef _VECTOR_H_
#define _VECTOR_H_

template<class T> class Vector // declaración
{
private:
    T *vector; // puntero al primer elemento de la matriz
    int nElementos; // número de elementos de la matriz
protected:
    T *asignarMem(int); // asignar memoria para vector
public:
    Vector(int ne = 10); // crea un Vector con ne elementos
    Vector(const Vector&); // crea un Vector desde otro
    ~Vector() { delete [] vector; vector = 0; } // destructor
    Vector& operator=(const Vector&); // operador de asignación
    T& operator[](int i) { return vector[i]; } // indexación
    int longitud() const { return nElementos; } // n° de elementos
};

#include "vector.cpp" // definición

#endif // _VECTOR_H_

```

Vamos a escribir una versión de esta plantilla para punteros a **void**; esta versión recibe el nombre de *especialización explícita*. Posteriormente utilizaremos esta versión como interfaz común para todas las matrices de punteros:

```
// vectorEspec.h - Especialización explícita de la plantilla Vector
#ifndef _VECTORESPEC_H_
#define _VECTORESPEC_H_
#include "vector.h"

template<> class Vector<void *> // declaración
{
private:
    void **vector; // puntero a la matriz de punteros
    int nElementos; // número de elementos de la matriz
protected:
    void **asignarMem(int); // asignar memoria para vector
public:
    Vector<void *>(int ne = 10);
    Vector<void *>(const Vector<void *>&);
    ~Vector<void *>() { delete [] vector; vector = 0; }
    Vector<void *>& operator=(const Vector<void *>&);
    void *& operator[](int i) { return vector[i]; }
    int longitud() const { return nElementos; }
};

#include "vectorEspec.cpp" // definición

#endif // _VECTORESPEC_H_
```

El prefijo `template<>` indica que la definición es una *especialización explícita* de la plantilla. Los valores de los parámetros para los que se utilizará la plantilla se especifican de forma explícita a continuación del nombre entre `<>`; es decir, `<void *>` indica que esta plantilla especializada se va a utilizar para generar una clase `Vector<void *>` cuando el valor pasado en la llamada para el parámetro *T* sea **void ***.

Las definiciones de los miembros de una especialización explícita de una plantilla de clase son definidos de la misma forma que los miembros de las clases normales, por lo que no utilizan la sintaxis `template<> declaración` que define una especialización explícita:

```
// vectorEspec.cpp - Especialización explícita de la plantilla
#include <iostream>
using namespace std;

// Crear una matriz con ne elementos
Vector<void *>::Vector<void *>(int ne)
{
```

```

    if (ne < 1)
    {
        cerr << "Nº de elementos no válido: " << ne << "\n";
        return;
    }
    nElementos = ne;
    vector = asignarMem(nElementos);
}

// Constructor copia
Vector<void *>::Vector<void *>(const Vector<void *>& v) : vector(0)
{
    *this = v;
}

// Operador de asignación
Vector<void *>& Vector<void *>::operator=(const Vector<void *>& v)
{
    nElementos = v.nElementos;           // número de elementos
    delete [] vector;                    // borrar la matriz actual
    vector = asignarMem(nElementos);     // crear una nueva matriz
    for (int i = 0; i < nElementos; i++)
        vector[i] = v.vector[i];        // copiar los valores
    return *this;                        // permitir asignaciones encadenadas
}

// Otros métodos
void **Vector<void *>::asignarMem(int nElems)
{
    try
    {
        void **p = new void *[nElems];
        return p;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        exit(-1);
    }
}

```

Esta especialización se utiliza para declarar vectores como el siguiente:

```
Vector<void *> v;
```

Evidentemente, en vez de haber escrito una plantilla explícita para puntero a **void**, podríamos haber escrito una clase con un nombre diferente (por ejemplo, *VectorPVoid*) pero perderíamos la flexibilidad de que fuera seleccionada automáticamente por el compilador basándose en los argumentos pasados a la plantilla,

de tal forma que la utilización de la declaración anterior en vez de *VectorPVoid* *v*, simplemente por olvido, acabaría utilizando la plantilla general *Vector<T>* que no está especializada para punteros.

Para poder declarar otros vectores como los siguientes:

```
Vector<double *> vector1;
Vector<string *> vector2;
```

hay que definir una *especialización parcial*, sólo para matrices de punteros, que pueda ser utilizada para cada tipo de puntero seleccionado. Como la interfaz común para matrices de punteros ya nos la proporciona la *especialización explícita* para punteros a **void** que acabamos de escribir, utilizar ésta con cualquier matriz de punteros supone simplemente derivar de ella otra versión de la plantilla *Vector* para punteros a *T*:

```
// vectorT.h - Especialización parcial de la plantilla Vector
#ifdef !defined( _VECTORT_H_ )
#define _VECTORT_H_

#include "vectorEspec.h"

template<class T> class Vector<T *> : private Vector<void *>
{
public:
    Vector(int ne = 10) : Vector<void *>(ne) {}
    T *& operator[](int i) { return reinterpret_cast<T *>(&
        Vector<void *>::operator[](i)); }
    int longitud() const { return Vector<void *>::longitud(); }
};

#endif // _VECTORT_H_
```

Obsérvese que cuando se utilice esta especialización (por ejemplo, con esta declaración, *Vector<string *> v*), el parámetro *T* se deduce del patrón de especialización *<T *>* especificado a continuación del nombre de la plantilla; en concreto, para el ejemplo anterior, *T* es **string** y no *string **.

Esta técnica de obtener una especialización parcial por derivación desde una especialización explícita es importante, porque favorece la contención de la explosión de código, en otras palabras, el código replicado es menor. Para verlo, sólo hay que comparar la definición de *Vector<void *>* con la definición de *Vector<T *>*; en ésta última no aparecen algunos métodos que sí están en *Vector<void *>*.

A continuación se muestra un ejemplo de trabajo con matrices de punteros. Este ejemplo crea primero una matriz de punteros a objetos **double** y después otra a objetos de tipo *t_nro_tfno*. El código pone a prueba los métodos implementados en las plantillas, tales como el constructor copia o los operadores de asignación e indexación, e implementa plantillas genéricas para visualizar y copiar una matriz de punteros a objetos **double**, así como una función para visualizar una matriz de punteros a objetos de tipo *t_nro_tfno*.

```
// test.cpp - Aplicación
#include <iostream>
#include <iomanip>
#include <string>
#include "vectorT.h"
using namespace std;

struct t_nro_tfno
{
    string nombre;
    long telefono;
};

void visualizar(Vector<t_nro_tfno *>& v)
{
    for (int i = 0; i < v.longitud(); i++)
    {
        cout << "Nombre:    " << v[i]->nombre << endl;
        cout << "Teléfono:  " << v[i]->telefono << endl;
    }
    cout << "\n\n";
}

template<class T> void visualizar(Vector<T *>& v)
{
    for (int i = 0; i < v.longitud(); i++)
        cout << setw(7) << *(v[i]);
    cout << "\n\n";
}

template<class T> Vector<T *> copiar(Vector<T *>& v)
{
    Vector<T *> x(v.longitud());
    for (int i = 0; i < v.longitud(); i++)
        x[i] = new T(*v[i]);
    return x;
}

int main()
{
    Vector<double *> vector1(3), vector2(3);
```

```

// Asignar datos a vector1
for (int i = 0; i < vector1.longitud(); i++)
    vector1[i] = new double(i+1);
// Visualizar los datos referenciados por vector1
cout << "vector1: "; visualizar(vector1);
// Hacer un duplicado de los datos: operador de asignación
vector2 = copiar(vector1);
// Poner a cero los datos referenciados por vector1
for (int i = 0; i < vector1.longitud(); i++)
    *(vector1[i]) = 0;
// Visualizar los datos referenciados por vector2 y vector1
cout << "vector2: "; visualizar(vector2);
cout << "vector1: "; visualizar(vector1);
// Liberar la memoria ocupada por los datos de vector1 y vector2
for (int i = 0; i < vector1.longitud(); i++)
{
    delete vector1[i];
    delete vector2[i];
}
// Matriz de elementos de tipo "t_nro_tfno"
Vector<t_nro_tfno *> vector3(5);
// Asignar datos
for (int i = 0; i < vector3.longitud(); i++)
{
    vector3[i] = new t_nro_tfno;
    cout << "Nombre:   "; getline(cin, vector3[i]->nombre);
    cout << "Teléfono: "; cin >> vector3[i]->telefono;
    cin.ignore();
}
// Visualizar los datos de vector3
cout << "vector3:\n"; visualizar(vector3);
// Liberar la memoria ocupada por los datos de vector3
for (int i = 0; i < vector1.longitud(); i++)
    delete vector3[i];
}

```

Derivación de plantillas

Una plantilla puede derivarse de otra plantilla o de una clase normal y construir así un nuevo tipo. La plantilla `Vector<T *>` implementada en el apartado anterior es un ejemplo de ello. Este ejemplo nos enseña que la derivación es la forma de utilizar una implementación común para un conjunto de plantillas.

Este otro ejemplo que exponemos a continuación implementa una plantilla `Matriz2d<T>` derivada de la plantilla general `Vector<T>`, con la intención de atender al tratamiento de las matrices de dos dimensiones de elementos de tipo `T`.

La forma de proceder no difiere de lo que ya aprendimos cuando estudiamos clases derivadas. A continuación se muestra la declaración y la definición de la plantilla *Matriz2d*. Se puede observar cómo ésta implementa sus métodos apoyándose en la funcionalidad heredada de la plantilla de la clase base.

La matriz de dos dimensiones a la que hemos hecho referencia está simulada por una matriz de punteros referenciada por el atributo *matriz2d* (que se inicia con la dirección del primer elemento de dicha matriz), cuyos elementos apuntan, a su vez, a otras matrices de tipo *T* que representan las filas. Su número de filas viene dado por el atributo *nElementos* de *Vector<T>* y su número de columnas por el atributo *nCols* de *Matriz2d<T>*. Se han implementado el constructor que invoca explícitamente al constructor de la base para pasarle el número de filas, el destructor que invoca implícitamente al destructor de la base, el operador función para acceder al elemento (*f*, *c*) de la matriz y dos métodos más que nos permiten obtener el número de filas y de columnas de la matriz. El constructor copia y el operador de asignación se implementan en el apartado *Ejercicios resueltos*.

```
// matriz2d.h - Plantilla Matriz2d derivada de Vector<T>
#ifndef _MATRIZ2D_H_
#define _MATRIZ2D_H_
#include "vector.h"

template<class T> class Matriz2d : public Vector<T> // declaración
{
private:
    T **matriz2d; // puntero al primer elemento de la matriz
    int nCols;    // número de columnas de la matriz
public:
    Matriz2d(int nf = 1, int nc = 1);
    ~Matriz2d();
    T& operator()(int f, int c); // operador función
    int filas() const { return longitud(); } // nº de filas
    int columnas() const { return nCols; } // nº de columnas
};

#include "matriz2d.cpp" // definición

#endif // _MATRIZ2D_H_

// matriz2d.cpp - Definición de la plantilla Matriz2d
#include <iostream>
using namespace std;

template<class T> Matriz2d<T>::Matriz2d(int nf, int nc) :
Vector<T>(nf)
{
    nCols = nc; // columnas
    matriz2d = reinterpret_cast<T **>(&(Vector<T>::operator[](0)));
```



```

    for (int f = 0; f < nf; f++)
        matriz2d[f] = asignarMem(nCols); // memoria para la fila f
}

template<class T> Matriz2d<T>::~Matriz2d()
{
    for (int f = 0; f < filas(); f++)
        delete [] matriz2d[f];
}

template<class T> T& Matriz2d<T>::operator()(int f, int c)
{
    return matriz2d[f][c];
}

```

El hecho de que las plantillas para la clase base y para la derivada tengan el mismo parámetro T es lo más común, pero no siempre es así.

La aplicación siguiente define una matriz de dos dimensiones $m2d$, la asigna datos desde el teclado y la visualiza utilizando una función genérica:

```

// test.cpp - Aplicación
#include <iostream>
#include <iomanip>
#include "matriz2d.h"
using namespace std;

template<class T> void visualizar(Matriz2d<T>& m)
{
    for (int f = 0; f < m.filas(); f++)
    {
        for (int c = 0; c < m.columnas(); c++)
            cout << setw(7) << m(f, c);
        cout << '\n';
    }
    cout << "\n\n";
}

int main()
{
    Matriz2d<double> m2d(2, 2);
    for (int f = 0; f < m2d.filas(); f++)
        for (int c = 0; c < m2d.columnas(); c++)
        {
            cout << "elemento[" << f << "][" << c << "] = ";
            cin >> m2d(f,c);
        }
    visualizar(m2d);
}

```

Otras características de las plantillas

Una plantilla de clase puede especificar valores por omisión para sus parámetros (los valores por omisión no pueden ser utilizados en plantillas de función). Por ejemplo, en la plantilla de clase *Matriz2d* podríamos utilizar un parámetro *d* de tipo **int** con un valor por omisión, para especificar el número de filas y columnas de una matriz cuadrada. También podríamos utilizar un parámetro *C* que nos permitiera acceder a la funcionalidad proporcionada por distintas plantillas de clase (*Cx<T>*, *Cy<T>*, etc.) y utilizar en la definición de *Matriz2d* una u otra según nuestras necesidades (por omisión siempre utilizaríamos una); el acceso a esta funcionalidad sería según la siguiente sintaxis *C::metodo()*.

En el código que se muestra a continuación es importante fijarse, desde un punto de vista sintáctico, dónde tienen que aparecer los parámetros de la plantilla.

```
template<class T> class Cx {};
```

```
// Declaración de la plantilla Matriz2d<T,d,C>
template<class T, int d = 2, class C = Cx<T> > class Matriz2d :
public Vector<T>
{
    // ...
    public:
        Matriz2d(int nf = d, int nc = d);
        Matriz2d(const Matriz2d<T,d,C>&); // constructor copia
        ~Matriz2d();
        // ...
};

template<class T, int d , class C> Matriz2d<T,d,C>::~~Matriz2d()
{
    for (int f = 0; f < filas(); f++)
        delete [] matriz2d[f];
}
// ...

// Funciones externas
template<class T, int d> void visualizar(Matriz2d<T,d>& m)
{
    for (int f = 0; f < d; f++)
    {
        for (int c = 0; c < d; c++)
            cout << setw(7) << m(f, c);
        cout << '\n';
    }
    cout << "\n\n";
}
```

```
int main()
{
    Matriz2d<double> m1(2, 2);
    Matriz2d<double, 2> m2;
    Matriz2d<double, 2, Cx<double> > m2d, v1;
    // ...
}
```

También, el estándar de C++ permite que una clase normal incluya miembros que sean plantillas, que las plantillas de clase contengan declaraciones de clases anidadas, etc. En estos casos resulta más sencillo escribir las definiciones en la propia declaración de la clase o de la plantilla de clase que fuera de ella. A continuación se muestra un ejemplo que aclara lo expuesto.

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

class Cx
{
public:
    template<class T> T menor(T a, T b)
    {
        return a < b ? a : b;
    }
};

template<class T1 = int, class T2 = T1 *> class Cy
{
public:
    class Cz
    {
    };

    template<int K = 3> class Cx2
    {
public:
        template<class T3> static T2 f(T3 n)
        {
            return new T1[n*K];
        }
    };
};

int main()
{
    Cx obx;
    Cy<int> oby;
    Cy<double>::Cx2<> obx2;

    int m = obx.menor(6, 3);
}
```

```
double *a = obx2.f<int>(2);
// ...
delete [] a;
}
```

A continuación se muestra cómo escribir la definición de la plantilla *Cy* fuera de la declaración. Comprobará que resulta bastante más complejo, razón por la que, en estos casos, se aconseja proceder como se ha indicado anteriormente.

```
template<class T1 = int, class T2 = T1 *> class Cy
{
public:
    class Cz;
    template<int K = 3> class Cx2;
};

template<class T1, class T2> class Cy<T1,T2>::Cz
{
};

template<class T1, class T2>
template<int K> class Cy<T1,T2>::Cx2
{
public:
    template<class T3> static T2 f(T3);
};

template<class T1, class T2>
template<int K>
template<class T3>
T2 Cy<T1,T2>::Cx2<K>::f(T3 n)
{
    return new T1[n*K];
}
```

Así mismo, en ocasiones, el empleo de **typedef** puede resultar más cómodo a la hora de trabajar con clases genéricas. Por ejemplo, la biblioteca estándar define **string** como un sinónimo:

```
typedef basic_string<char> string;
```

Finalmente, conviene recordar que la biblioteca denominada *STL* está formada por un conjunto de plantillas con el único objetivo de ofrecer algoritmos eficientes y genéricos. Sirvan como ejemplo los contenedores **vector** y **map**.

EJERCICIOS RESUELTOS

1. En el apartado *Derivación de plantillas* escribimos una plantilla *Matriz2d* derivada de *Vector*. Como ejercicio, completar dicha plantilla añadiendo el constructor copia y el operador de asignación. Finalmente, realizar un ejemplo que pruebe ambos métodos.

```
// matriz2d.h - Plantilla Matriz2d derivada de Vector<T>
#ifndef _MATRIZ2D_H_
#define _MATRIZ2D_H_
#include "vector.h"

template<class T> class Matriz2d : public Vector<T> // declaración
{
private:
    T **matriz2d; // puntero al primer elemento de la matriz
    int nCols; // número de columnas de la matriz

public:
    Matriz2d(int nf = 1, int nc = 1);
    Matriz2d(const Matriz2d<T>&); // constructor copia
    ~Matriz2d();
    void destruir();
    Matriz2d& operator=(const Matriz2d<T>&); // operador =
    T& operator()(int f, int c); // operador ()
    int filas() const { return longitud(); } // nº de filas
    int columnas() const { return nCols; } // nº de columnas
};

// matriz2d.cpp - Definición de la plantilla Matriz2d
#include <iostream>
using namespace std;

template<class T> Matriz2d<T>::Matriz2d(int nf, int nc) :
Vector<T>(nf)
{
    nCols = nc; // columnas
    matriz2d = reinterpret_cast<T **>(&(Vector<T>::operator[](0)));
    for (int f = 0; f < nf; f++)
        matriz2d[f] = asignarMem(nCols); // memoria para la fila f
}

template<class T> Matriz2d<T>::~Matriz2d()
{
    destruir();
}

template<class T> void Matriz2d<T>::destruir()
{

```

```
        for (int f = 0; f < filas(); f++)
            delete [] matriz2d[f];
    }

template<class T> T& Matriz2d<T>::operator()(int f, int c)
{
    return matriz2d[f][c]; // invoca a Vector<T>::operator[]
}

// Constructor copia
template<class T> Matriz2d<T>::Matriz2d(const Matriz2d<T>& v) :
matriz2d(0)
{
    *this = v;
}

// Operador de asignación
template<class T>
Matriz2d<T>& Matriz2d<T>::operator=(const Matriz2d<T>& v)
{
    if (matriz2d)
    {
        if (v.filas() != filas() || v.columnas() != columnas())
        {
            cout << "Las matrices no tienen las mismas dimensiones\n";
            return *this;
        }
        destruir(); // borrar la matriz actual.
    }
    Vector<T>::operator=(v); // operador = de la base
    nCols = v.nCols;
    // Asignar a matriz2d la dirección del primer elemento de la
    // matriz de punteros
    matriz2d = reinterpret_cast<T **>(&(Vector<T>::operator[](0)));
    for (int f = 0; f < filas(); f++)
    {
        matriz2d[f] = asignarMem(nCols); // asignar memoria
        for (int c = 0; c < nCols; c++)
            matriz2d[f][c] = v.matriz2d[f][c]; // copiar los valores
    }

    return *this;          // permitir asignaciones encadenadas
}
```

2. Una matriz multidimensional en C++ representa un conjunto de elementos sucesivos en memoria que pueden ser accedidos mediante variables suscritas o de subíndices. Dichos subíndices son especificados utilizando uno o más operadores []. Por ejemplo:

```
int miMatrizInt[5][10][4];
```

```
int i, j, k, conta = 1;
// ...
miMatrizInt[i][j][k] = conta++;
```

Esta misma construcción puede realizarse desde una programación orientada a objetos. Para ello, podemos definir una plantilla de clase que incluya, además, la verificación de que los subíndices están dentro de los límites establecidos para la matriz y utilizar esta plantilla de forma análoga a como se indica a continuación:

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

int main()
{
    const int A = 5;
    const int B = 10;
    const int C = 4;

    CMatriz<int> miMatrizInt(A, B, C);
    int i, j, k, conta = 1;

    for (i = 0; i < A; i++)
        for (j = 0; j < B; j++)
            for (k = 0; k < C; k++)
                miMatrizInt[i][j][k] = conta++;

    for (i = 0; i < A; i++)
        for (j = 0; j < B; j++)
            for (k = 0; k < C; k++)
                cout << miMatrizInt[i][j][k] << " ";
    cout << endl;

    CMatriz<double> miMatrizDouble(A, B);
    miMatrizDouble[1][2] = 7.5;
}
```

Como vemos en el ejemplo, la plantilla *CMatriz* puede utilizarse para declarar matrices de cualquier número de dimensiones (en nuestra implementación, máximo seis). Observamos también que la verificación de los subíndices cuando accedemos a un elemento de la matriz no se hace *in situ*, sino que queda pospuesta a la ejecución.

Para realizar lo expuesto, crearemos una plantilla de clase de la siguiente manera:

```
////////////////////////////////////
// Plantilla de clase para construir matrices multidimensionales
```

```
template<class T> class CElementoAccedido;
template<class T> class CMatriz
{
    friend class CElementoAccedido<T>;
    T *m_pMatriz; // matriz lineal de elementos de tipo T
    CDimensiones m_Dimensiones; // dimensiones de la matriz

public:
    // Constructor: matriz de una dimensión
    CMatriz(int s1) : m_Dimensiones(1, s1, 0, 0, 0, 0, 0)
    {
        m_pMatriz = new T[m_Dimensiones.TotalElementos()];
    }

    // Constructor: matriz de dos dimensiones
    CMatriz(int s1, int s2) : m_Dimensiones(2, s1, s2, 0, 0, 0, 0)
    {
        m_pMatriz = new T[m_Dimensiones.TotalElementos()];
    }

    // Constructor: matriz de tres dimensiones
    CMatriz(int s1, int s2, int s3) :
    m_Dimensiones(3, s1, s2, s3, 0, 0, 0)
    {
        m_pMatriz = new T[m_Dimensiones.TotalElementos()];
    }

    // Constructor: matriz de cuatro dimensiones
    CMatriz(int s1, int s2, int s3, int s4) :
    m_Dimensiones(4, s1, s2, s3, s4, 0, 0)
    {
        m_pMatriz = new T[m_Dimensiones.TotalElementos()];
    }

    // Constructor: matriz de cinco dimensiones
    CMatriz(int s1, int s2, int s3, int s4, int s5) :
    m_Dimensiones(5, s1, s2, s3, s4, s5, 0)
    {
        m_pMatriz = new T[m_Dimensiones.TotalElementos()];
    }

    // Constructor: matriz de seis dimensiones
    CMatriz(int s1, int s2, int s3, int s4, int s5, int s6) :
    m_Dimensiones(6, s1, s2, s3, s4, s5, s6)
    {
        m_pMatriz = new T[m_Dimensiones.TotalElementos()];
    }

    // Constructor copia
    CMatriz(const CMatriz<T> &x);
};
```



```

    T &ObtenerElemento(); // busca el elemento en el espacio físico

public:
    CElementoAccedido<T> &operator[](int subind)
    {
        m_Subindices.AnyadirSigSubind(subind);
        return *this;
    }

    //
    operator T &()
    {
        return ObtenerElemento();
    }

    // Asigna o devuelve el valor a/de un elemento de la matriz
    T &operator=(const T &arg) { return ObtenerElemento() = arg; }
};
/////////////////////////////////////////////////////////////////

```

El dato miembro *m_Matriz* es una referencia a la matriz multidimensional y *m_Subindices* es un objeto de la clase *CSubindices* que básicamente almacena una matriz de enteros correspondientes a los subíndices del elemento al que se accede (en el ejemplo, `miMatrizDouble[1][2] = 7.5, 1 y 2`) y el número total de subíndices (en el ejemplo, 2).

```

/////////////////////////////////////////////////////////////////
// Clase para acumular los subíndices. Básicamente crea una lista
// con los subíndices de un elemento determinado, [][][]
class CSubindices
{
    int *m_pSubindices; // matriz para los subíndices
    int m_nSubindices; // número total de subíndices
    int *m_pSiguieteInd; // localización del siguiente subíndice
    int m_nSubindsActual; // subíndices que hay actualmente

public:
    CSubindices(int nTotalSubinds, int PrimerSubind);
    ~CSubindices() { delete [] m_pSubindices; }
    int nSubindsActual() { return m_nSubindsActual; }
    void AnyadirSigSubind(int subind);
    int Desplazamiento(CDimensiones &dims) const;
};
/////////////////////////////////////////////////////////////////

```

Los otros dos datos miembros, *m_pSiguieteInd* y *m_nSubindsActual*, participan en la construcción de un objeto *CSubindices*. Fíjese en el constructor, recibe como argumentos el número total de subíndices necesarios para acceder a un elemento de la matriz multidimensional y el primer subíndice de éstos. Por lo tanto,

inicialmente *m_pSiguieteInd* tendrá el mismo valor que *m_pSubindices* y *m_nSubindsActual* tendrá el valor 1. El método *AnyadirSigSubind* añadirá el resto de los subíndices a la matriz referenciada por *m_pSubindices*, el cual recibe como parámetro el siguiente subíndice del elemento al que estamos accediendo. El método *Desplazamiento* calcula la posición dentro de la matriz lineal (de una dimensión) equivalente a la posición dentro de la matriz multidimensional, indicada por los subíndices.

```

////////////////////////////////////
// Implementación de los métodos
// Buscar el elemento en el espacio físico (matriz lineal)
template<class T> T &CElementoAccedido<T>::ObtenerElemento()
{
    static T nada;

    // El número de dimensiones tiene que ser igual al número de
    // subíndices
    if (m_Matriz.m_Dimensiones.nDimensiones() !=
        m_Subindices.nSubindsActual())
    {
        cerr << "Número de dimensiones y subíndices diferentes\n";
        return nada;
    }
    // Calcular el desplazamiento en el espacio físico de la matriz
    // correspondiente a los subíndices del elemento elegido. También
    // se verifica la validez de los subíndices.
    int desplazamiento =
        m_Subindices.Desplazamiento(m_Matriz.m_Dimensiones);
    if (desplazamiento != -1)
        return m_Matriz.m_pMatriz[desplazamiento];
    else
        return nada;
}

// Constructor
CSubindices::CSubindices(int nTotalSubinds, int PrimerSubind)
{
    if (nTotalSubinds < 1)
    {
        cerr << "Número de subíndices incorrecto\n";
        nTotalSubinds = 1;
        PrimerSubind = 0;
    }
    m_pSubindices = m_pSiguieteInd =
        new int[m_nSubindices=nTotalSubinds];
    *m_pSiguieteInd++ = PrimerSubind;
    m_nSubindsActual = 1;
}

```

```

// Añadir el siguiente subíndice de un elemento a su matriz de
// subíndices
void CSubindices::AnyadirSigSubind(int subind)
{
    if (m_nSubindsActual < m_nSubindices)
    {
        *m_pSiguienteInd++ = subind;
        m_nSubindsActual++;
    }
}

// Calcular el desplazamiento
int CSubindices::Desplazamiento(CDimensiones &dims) const
{
    int desplazamiento = 0; // resultado
    int *pSubind = m_pSubindices; // subíndices del elemento accedido
    const int *dimension = dims.DimensionesMatriz(); // dimensiones
    int nDims = dims.nDimensiones();

    while (nDims--)
    {
        // Verificar si los subíndices están dentro del rango
        if (*pSubind < 0 || *pSubind >= *dimension)
        {
            cerr << "Subíndice fuera de rango\n";
            return -1;
        }
        desplazamiento += *pSubind++;
        if (nDims) desplazamiento *= *++dimension;
    }
    return desplazamiento;
}

// Construir un objeto CDimensiones para contener cada una de las
// dimensiones y el número de ellas.
void CDimensiones::Construccion(int nDims, int *pDims)
{
    int *dim = pDims;
    for (int i = nDims; i--; dim++)
        if (*dim < 1)
        {
            cerr << "Dimensión nula o negativa\n";
            // Asignar valores por omisión
            m_DimensionesMatriz = new int[m_nDimensiones = 1];
            m_DimensionesMatriz[0] = 1;
            return;
        }
    m_DimensionesMatriz = new int[m_nDimensiones = nDims];
    memcpy(m_DimensionesMatriz, pDims, nDims * sizeof(int));
}

```

```
// Total de elementos de la matriz multidimensional
int CDimensiones::TotalElementos() const
{
    int nElementos = m_DimensionesMatriz[0]; // total elementos
    int nDims = m_nDimensiones; // contador
    int *dim = m_DimensionesMatriz + 1; // siguiente dimensión
    while (--nDims) nElementos *= *dim++;
    return nElementos;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Analice el código anterior e intente construir un programa ejecutable utilizando la función **main** expuesta al principio de este ejercicio. Después, y antes de ejecutar el programa, intente responder a las preguntas del siguiente apartado. Para contrastar la solución puede ejecutar el programa *matriz.cpp* incluido en la carpeta ...*\Ejemplos del libro\cap14\matriz* del CD que acompaña al libro.

EJERCICIOS PROPUESTOS

Utilizando la funcionalidad de las clases descritas anteriormente, responda brevemente a las siguientes preguntas:

1. Sustituya los constructores actuales de *CMatriz<T>* por un único constructor. Recuerde que dicho constructor era ejecutado por declaraciones como la siguiente:

```
CMatriz<int> miMatrizInt(4, 8, 5); // matriz de tres dimensiones
```

Como ve sólo se especifican las dimensiones, sin especificar el número de ellas. Siguiendo este mismo criterio, escriba el nuevo constructor dentro de la declaración de la clase, esto es:

```
template<class T> class CMatriz
{
    // ...
};
```

2. ¿La definición del nuevo constructor modifica en algo el resto del código? En caso afirmativo diga qué y realice las modificaciones.
3. Escriba el código que falta en el siguiente método de la clase *CMatriz<T>*:

```
CElementoAccedido<T> operator[]( int subind )
{
    return
}
```

4. Escriba los prototipos de los métodos que son llamados, pertenecientes a las clases descritas, y en el orden en el que son llamados cuando se ejecuta la declaración siguiente:

```
CMatriz<float> miMatrizFloat(A, B);
```

5. El método de la clase *CElementoAccedido<T>*:

```
operator T &()
{
    return ((CElementoAccedido<T> *)this)->ObtenerElemento();
}
```

¿Cómo se denomina? ¿Qué valor retorna? ¿Para qué sirve?

6. Suponiendo que ha declarado la matriz *miMatrizFloat* especificada en el apartado 4, escriba una sentencia que provoque la llamada, entre otras, del método indicado a continuación correspondiente a la clase *CElementoAccedido*:

```
operator T &()
{
    return ((CElementoAccedido<T> *)this)->ObtenerElemento();
}
```

7. Escriba los prototipos de los métodos que son llamados, pertenecientes a las clases descritas, y en el orden en el que son llamados cuando se ejecuta la declaración siguiente:

```
cout << miMatrizFloat[1][2] << endl;
```

Tenga presente que la asociatividad del operador de indexación, [], es de izquierda a derecha (evalúe los operadores de uno en uno).

8. Escriba los nombres de las clases de objetos que son creados y en el orden en el que son creados cuando se ejecuta la declaración siguiente:

```
miMatrizDouble[1][2][3] = 5.5;
```

9. Cuando se ejecuta la declaración siguiente, ¿cuántos operadores de indexación son llamados? Escriba sus prototipos (*miMatrizDouble* es de tipo **double**).

```
miMatrizDouble[1][2][3] = 8.5;
```

10. Escriba el operador de asignación de la clase *CDimensiones* (vea el método *Construccion* de la clase *CDimensiones*) y el operador de asignación de la

clase *CMatriz*<*T*>. Escriba ambos operadores fuera de la declaración de la clase como métodos **inline**; esto es, fuera de las {}.

11. Escriba la llamada explícita correspondiente a la declaración siguiente (*miMatrizDouble* y *d* son de tipo **double**):

```
d = miMatrizDouble[2][3];
```

12. Partiendo de la siguiente declaración:

```
CMatriz<double> miMatrizDouble(A,B,C), OtraMatrizDouble(A,B,C);
```

la sentencia siguiente, ¿a qué método llama?

```
OtraMatrizDouble = miMatrizDouble;
```


EXCEPCIONES

El lenguaje C++ incorpora soporte para manejar situaciones anómalas, conocidas como “excepciones”, que pueden ocurrir durante la ejecución de un programa. Con el sistema de manipulación de excepciones de C++, un programa puede comunicar eventos inesperados a un contexto de ejecución más capacitado para responder a tales eventos anormales. Estas excepciones son manejadas por código fuera del flujo normal de control del programa.

Las excepciones proporcionan una manera limpia de verificar errores; esto es, sin abarrotar el código básico de una aplicación utilizando sistemáticamente los códigos de retorno de los métodos en sentencias **if** y **switch** para controlar los posibles errores que se puedan dar. Veamos, con un ejemplo, a qué nos estamos refiriendo:

```
int codidoDeError = 0;
codidoDeError = leerFichero(nombre);
if (codidoDeError != 0)
{
    // Ocurrió un error al leer el fichero
    switch (codidoDeError)
    {
        case 1:
            // No se encontró el fichero
            // ...
            break;
        case 2:
            // El fichero está corrupto
            // ...
            break;
        case 3:
            // El dispositivo no está listo
            // ...
    }
}
```

```
        break;
    default:
        // Otro error
        // ...
    }
}
else
{
    // Procesar los datos leídos del fichero
}
```

El código del ejemplo anterior trata de leer un fichero almacenado en el disco invocando al método *leerFichero*. Este método devuelve un valor 0 si se ejecuta satisfactoriamente y un valor distinto de 0 en otro caso. Para analizar este hecho se ha utilizado una sentencia **if**. En el caso de que se produzca un error, una sentencia **switch** se encargará de verificar qué es lo que ha ocurrido y de tratar de resolverlo de la mejor forma posible. Lo que se persigue es que el programa no sea abortado inesperadamente por el sistema, sino diseñar una continuación o terminación normal dentro de lo ocurrido.

Observemos el código que ha sido necesario escribir para tratar un posible error debido a no poder leer un fichero del disco. Pensemos, ¿cuántos errores más podrían abortar nuestra aplicación? Para que esto no suceda, ¿se imagina la complejidad del código escrito una vez añadido todo el necesario para tratar cada uno de ellos? El manejo de excepciones ofrece una forma de separar explícitamente el código que maneja los errores del código básico de una aplicación, haciéndola más legible, lo que desemboca en un buen estilo de programación. Por ejemplo:

```
try
{
    // Código de la aplicación
}
catch(clase_de_excepción e)
{
    // Código de tratamiento de esta excepción
}
catch(otra_clase_de_excepción e)
{
    // Código de tratamiento para otra clase de excepción
}
```

Básicamente, el esquema anterior dice que si el código de la aplicación no puede realizar alguna operación, se espera lance una excepción que será tratada por el código de tratamiento especificado para esa clase de excepción, o en su defecto por C++.

A lo largo de este capítulo comprobará que: el manejo de excepciones reduce la complejidad de la programación; los métodos que invocan a otros no necesitan comprobar valores de retorno; si el método invocado finaliza de forma normal, el que llamó está seguro de que no ocurrió ninguna situación anómala; etc.

EXCEPCIONES DE C++

Durante el estudio de los capítulos anteriores, seguro que se habrá encontrado con excepciones como las siguientes:

Clase de excepción	Cabecera	Significado
bad_alloc	<code><new></code>	Excepción lanzada por new cuando no hay memoria suficiente para asignación.
bad_cast	<code><typeinfo></code>	Excepción lanzada por dynamic_cast cuando no es posible realizar una conversión entre referencias.
bad_typeid	<code><typeinfo></code>	Excepción lanzada por typeid cuando su argumento es 0 o una dirección no válida.
bad_exception	<code><exception></code>	Excepción lanzada cuando ocurre un error no esperado.

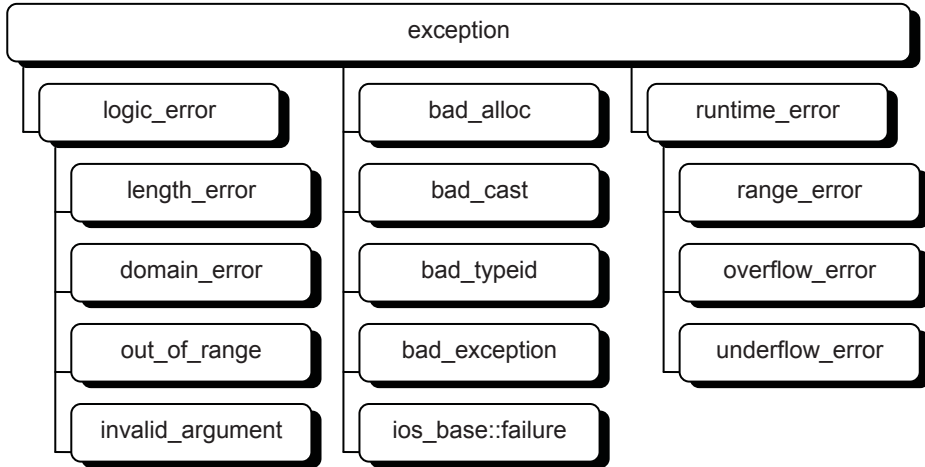
Éstas son excepciones estándar lanzadas por el lenguaje. Hay otras excepciones que son lanzadas por la biblioteca estándar, como, por ejemplo:

Clase de excepción	Cabecera	Significado
out_of_range	<code><stdexcept></code>	Excepción lanzada para informar que el valor de un argumento está fuera de rango.
invalid_argument	<code><stdexcept></code>	Excepción lanzada para informar que un argumento no es válido.
overflow_error	<code><stdexcept></code>	Excepción lanzada para informar de un desbordamiento aritmético.
ios_base::failure	<code><ios></code>	Excepción lanzada por el método clear de la plantilla de clase basic_ios .

¿Qué es lo que ocurrió entonces cuando durante la ejecución de su programa se lanzó una excepción? Seguramente el programa dejó de funcionar y C++ visualizó algún mensaje acerca de lo ocurrido. Si no es esto lo que deseamos, tendremos que aprender a manipular las excepciones.

Las excepciones en C++ son objetos de clases derivadas de la clase **exception** definida en el espacio de nombres **std**, o bien de otras clases definidas por nosotros. Por ejemplo, cuando se lanza una excepción **overflow_error**, automática-

mente C++ crea un objeto de esta clase. La figura siguiente muestra algunas de las clases de la jerarquía de excepciones de la biblioteca de C++:



La clase **exception** cubre las excepciones estándar que una aplicación normal puede manipular. Tiene varias clases derivadas entre las que destacan **logic_error**, **runtime_error**, **bad_alloc**, **bad_cast** y **bad_exception**. Entre los métodos que proporciona la clase **exception** destaca **what**, método virtual que devuelve un valor de tipo **const char *** relacionado con el error ocurrido.

Las clases derivadas de **exception** no añaden nueva funcionalidad, sino que simplemente definen los métodos virtuales necesarios de la manera apropiada. No obstante, no todas las excepciones tienen que derivarse de **exception**; de hecho, los desarrolladores de software suelen añadir su propia jerarquía de excepciones.

Un objeto **logic_error**, error lógico, es un error que en principio podría detectarse antes de que el programa comenzase a ejecutarse. Los demás son errores detectables durante la ejecución.

La clase **runtime_error** cubre las excepciones ocurridas al ejecutar operaciones sobre los datos que manipula la aplicación y que residen en memoria; se trata de excepciones que se lanzan durante la ejecución.

La clase **basic_ios** tiene un miembro **exceptions** que permite solicitar a **clear** que eleve una excepción **ios_base::failure** si después de una operación sobre un flujo de E/S ocurre un error (**clear** es el único método que modifica el estado de un flujo); esto evita tener que realizar comprobaciones, para detectar si ocurrió un error, después de cada operación de E/S.

Como ejemplo, recuerde que en las aplicaciones desarrolladas hasta ahora el compilador C++ nunca nos obligó a manejar una excepción de la clase **bad_alloc**, pero sí es conveniente hacerlo por si no hubiera memoria suficiente para asignación. Un ejemplo lo podemos ver a continuación:

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

class Cx
{
private:
    int a[25];
public:
    Cx() {}
    ~Cx() {}
    // ...
};

int main()
{
    Cx *p = 0;
    int n = 10;
    try
    {
        p = new Cx[n];
        // ...
    }
    catch(bad_alloc& e)
    {
        cout << e.what() << endl;
        return -1;
    }
    // ...
    delete [] p;
}
```

En este ejemplo se puede observar un bloque **try** que encierra el código que puede lanzar una excepción durante su ejecución y un bloque **catch** que capturaría esa excepción si fuera de la clase **bad_alloc**. Precisamente, ésta es la clase de excepción que lanzará **new** si no hubiera memoria suficiente para asignación.

No se preocupe si no le ha quedado todo claro; a continuación aprenderá con detalle cómo capturar, crear y lanzar excepciones, además de otras cosas.

MANEJAR EXCEPCIONES

Cuando un método se encuentra con una anomalía que no puede resolver, lo lógico es que *lance* (**throw**) una excepción, esperando que quien lo llamó directa o indirectamente la *capture* (**catch**) y maneje la anomalía. Incluso él mismo podría capturar y manipular dicha excepción. Si la excepción no se captura, el programa finalizará automáticamente.

Por ejemplo, ¿recuerda el flujo **cin**? El estado de este flujo, igual que el de cualquier otro, es modificado por el método **clear** siempre que una operación de entrada no tiene éxito. Por ejemplo, si utilizamos **cin** para obtener de la entrada estándar un dato **double** y se encuentra con un **string**, **clear** pondrá la bandera de estado **failbit** a uno para notificar que la entrada no es la esperada. Además de esto, podríamos solicitar a **clear** que lanzara una excepción de la clase **failure** invocando al método **exceptions** así:

```
cin.exceptions(ios::failbit | ios::badbit);
```

Según lo expuesto, podríamos escribir un método *leerDouble* como se indica a continuación. Este método invoca al operador de extracción con el propósito de devolver un **double** correspondiente al valor leído. Según se ha explicado anteriormente, si la entrada no es la esperada, **clear** puede lanzar una excepción de la clase **failure**. Para manejarla hay que capturarla, para lo cual se utiliza un bloque **catch**, y para poder capturarla hay que encerrar el código que puede lanzarla en un bloque **try**:

```
double leerDouble()
{
    cin.exceptions(ios::failbit | ios::badbit);
    double dato = 0.0;
    try
    {
        cin >> dato;
    }
    catch(ios_base::failure& e)
    {
        cout << e.what() << ": dato no válido\n";
        cin.clear();
        cin.ignore(numeric_limits<int>::max(), '\n');
    }
    return dato; // devolver el dato tecleado
}
```

Las palabras **try** y **catch** trabajan conjuntamente y pueden traducirse así: “poner a prueba un fragmento de código por si lanzara una excepción; si se ejecuta

satisfactoriamente, continuar con la ejecución del programa; si no, capturar la excepción lanzada y manejarla”.

Lanzar una excepción

Lanzar una excepción equivale a crear un objeto de la clase de la excepción para manipularlo fuera del flujo normal de ejecución del programa. Para lanzar una excepción se utiliza la palabra reservada **throw**. Por ejemplo, volviendo al método *leerDouble* expuesto anteriormente, si ocurre un error cuando se ejecute el operador **>>** sobre **cin**, se supone que éste a través de **clear** ejecutará una sentencia similar a la siguiente:

```
if (error) throw ios_base::failure();
```

Esta sentencia lanza una excepción de la clase **ios_base::failure** lo que implica crear un objeto de esta clase (*ios_base::failure()* es una llamada al constructor sin parámetros de esa clase, suponiendo que existe). Un objeto de éstos contiene información acerca de la excepción.

Se pueden lanzar excepciones de cualquier clase, incluso de los tipos primitivos, como **int**, o derivados, como **char ***. Por ejemplo:

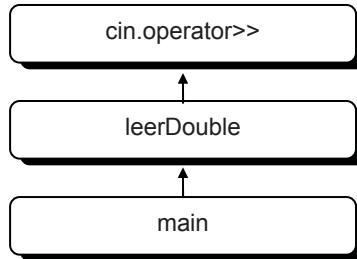
```
int idato = 10;
char *sdato = "error x";
// ...
    throw idato; // el manejador recibirá como argumento idato
// ...
    throw sdato; // el manejador recibirá como argumento sdato
```

Capturar una excepción

Una vez lanzada la excepción, el sistema es responsable de encontrar a alguien que la capture con el objetivo de manipularla. El conjunto de esos “alguien” es el conjunto de métodos especificados en la pila de llamadas hasta que ocurrió el error. Por ejemplo, consideremos la siguiente función, que invoca al método *leerDouble* con la intención de leer un dato:

```
int main()
{
    double d = 0;
    cout << "dato: "; d = leerDouble();
    // ...
}
```

Cuando se ejecute esta función y se invoque al método *leerDouble*, la pila de llamadas crecerá como se observa en la figura siguiente:



Si al ejecutarse el método **cin.operator>>** ocurriera un error, según hemos visto anteriormente, éste (a través del método **clear**) lanzaría una excepción de la clase **failure** que interrumpirá el flujo normal de ejecución. Después, el sistema buscaría en la pila de llamadas hacia abajo y comenzando por el propio método que produjo el error, uno que implemente un manejador que pueda capturar esta excepción. Si el sistema, descendiendo por la pila de llamadas, no encontrara este manejador, el programa terminaría.

Para implementar un manejador para una clase de excepción hay que hacer las dos cosas que se indican a continuación:

1. Encerrar el código que puede lanzar la excepción en un bloque **try**. El método *leerDouble* tiene un bloque **try** que encierra la llamada al método **operator>>** (además de a esta sentencia, podría encerrar a otras):

```

try
{
    cin >> dato;
}
  
```

2. Escribir un bloque **catch** capaz de capturar la excepción lanzada. El método *leerDouble* tiene un bloque **catch** capaz de capturar excepciones de la clase **ios_base::failure** y de sus subclases, si las hubiera:

```

catch(ios_base::failure& e)
{
    cout << e.what() << ": dato no válido\n";
    cin.clear();
    cin.ignore(numeric_limits<int>::max(), '\n');
}
  
```

En este manejador se observa un parámetro *e* que referencia al objeto que se creó cuando se lanzó la excepción capturada. Para manipularla, además de escribir

el código que consideremos adecuado, disponemos de la funcionalidad proporcionada por la clase **failure**, y a la que podremos acceder mediante el objeto *e*. Por ejemplo, el método **what** devuelve una cadena con información acerca de la excepción ocurrida.

Un manejador de excepción, **catch**, sólo se puede utilizar a continuación de un bloque **try** o de otro manejador de excepción (bloque **catch**). Las palabras clave **try** y **catch**, por definición, van seguidas de un bloque que encierra el código relativo a cada una de ellas, razón por la cual es obligatorio utilizar llaves: {}.

El siguiente ejemplo muestra un manejador para una excepción de tipo **int** y otro para una excepción de tipo **char ***:

```
catch(int e)
{
    // e es un entero pasado por una sentencia como "throw idato"
}
catch(char *e)
{
    // e es un cadena pasada por una sentencia como "throw sdato"
}
```

Cuando una excepción es capturada se considera manejada, lo que significa que cualquier otro manejador existente no será tenido en cuenta.

Excepciones derivadas

Cuando se trata de manejar excepciones, un bloque **try** puede estar seguido de uno o más bloques **catch**, tantos como excepciones diferentes tengamos que manejar. Cada **catch** tendrá un parámetro de la clase **exception**, de alguna clase derivada de ésta o bien de una clase de excepción definida por el usuario. Cuando se lance una excepción, el bloque **catch** que la capture será aquél cuyo parámetro sea de la clase de la excepción o de una clase base directa o indirecta. Debido a esto, el orden en el que se coloquen los bloques **catch** tiene que ser tal, que cualquiera de ellos debe permitir alcanzar el siguiente, de lo contrario habrá bloques que nunca se ejecuten. Generalmente, el compilador suele avisar de este hecho.

Por ejemplo, si el primer bloque **catch** especifica un parámetro de la clase **exception**, ningún otro bloque que le siga con un parámetro de alguna de sus derivadas podría alcanzarse; esto es, cualquier excepción lanzada sería capturada por ese primer bloque, ya que cualquier objeto, puntero o referencia a una clase puede ser convertido implícitamente por C++ en un objeto, puntero o referencia a su clase base directa o indirecta.

En cambio, en el ejemplo siguiente, una excepción de la clase **out_of_range** será capturada por el primer bloque **catch**; una excepción de la clase **logic_error** será capturada por el bloque segundo; una excepción de la clase **invalid_argument**, derivada de **logic_error**, será capturada también por el bloque segundo; y una excepción de la clase **bad_alloc**, subclase de **exception**, será capturada por el bloque tercero.

```
try
{
    // ...
}
catch(out_of_range& e)
{
    // Manejar esta clase de excepción
}
catch(logic_error& e)
{
    // Manejar esta clase de excepción o de alguna de sus derivadas,
    // excepto out_of_range
}
catch(exception& e)
{
    // Manejar esta clase de excepción o de alguna de sus derivadas,
    // excepto out_of_range y logic_error
}
```

Recordar también que, cuando trabajamos con una jerarquía de excepciones en la que existan clases polimórficas, capturar una excepción por referencia permitirá que el método virtual invocado pertenezca a la clase del objeto referenciado.

Capturar cualquier excepción

Tres puntos suspensivos **...** indican cualquier parámetro, por lo que *catch(...)* significa “capturar cualquier excepción”. Por ejemplo, al código del ejemplo anterior le podríamos añadir un último bloque **catch** que capture todas las excepciones con la intención de realizar una finalización adecuada del proceso que estaba en curso, e incluso podríamos a continuación relanzar la excepción para que fuera tratada por otro manejador.

```
// ...

catch(exception& e)
{
    // Manejar esta clase de excepción o de alguna de sus derivadas,
    // excepto out_of_range y logic_error
}
```

```

catch(...)
{
    // Finalización adecuada para el proceso en curso.
    throw; // lanzar otra vez la excepción
}

```

Relanzar una excepción

Después de haber capturado una excepción, el manejador puede volverla a lanzar; esto se hace ejecutando **throw** sin ningún operando, según se puede observar en el ejemplo anterior. Esto puede ser útil en los casos en los que el manejador que capturó una excepción decida que no puede tratar completamente el error, por ejemplo, porque no está disponible en este lugar toda la información que necesita, en cuyo caso hace lo que puede y después vuelve a lanzar la excepción otra vez, con la intención de que el tratamiento pueda ser continuado por otro manejador.

CREAR EXCEPCIONES

En alguna ocasión puede que necesitemos crear nuestras propias excepciones, a pesar de que en la biblioteca de clases de C++ hay una gran cantidad de ellas que podemos utilizar sin más. Esos nuevos tipos de excepción no tienen por qué corresponderse con una clase derivada de **exception**, clase raíz de la jerarquía de clases de excepciones de C++. Más aún, muchos de los errores que ocurren en la mayoría de los programas no se corresponden con excepciones de alguna de las subclases de **exception**.

En general, crearemos un nuevo tipo de excepción cuando queramos manejar un determinado tipo de error no contemplado por las excepciones proporcionadas por la biblioteca de C++. Por ejemplo, para crear un tipo de excepción *EValorNoValido*, con la intención de manejar un error “valor no válido”, podemos diseñar una clase como la siguiente:

```

class EValorNoValido
{
    string mensajeDeError;
public:
    EValorNoValido(string mensaje = "valor no válido")
    {
        mensajeDeError = mensaje;
    }
    string what() { return mensajeDeError; }
    // ...
};

```

Según se observa en este ejemplo, la nueva clase de excepción *EValorNoValido* implementa un constructor con un parámetro de tipo **string** que tiene asignado un valor por omisión. El parámetro de tipo **string** es el mensaje que devolverá el método **what**, método análogo al proporcionado por la clase **exception**.

La clase de excepción *EValorNoValido* relacionada con el error “valor no válido” ya está creada. Lógicamente, siempre que se implemente una clase de excepción es porque, durante el desarrollo de un proyecto cualquiera, se observa que su código para determinados valores durante la ejecución puede presentar una anomalía de la que los usuarios deben ser informados con el fin de que la puedan tratar. Para ello, habrá entonces que añadir el código que verifique si se producen esos valores y en caso afirmativo lanzar la excepción programada para este caso. Por ejemplo:

```
void f(double a)
{
    if (a == 0)
        throw EValorNoValido("Error: valor cero");
    // ...
}
```

Según hemos estudiado anteriormente, lanzar una excepción equivale a crear un objeto de ese tipo de excepción. En el ejemplo anterior se observa que la circunstancia que provoca el error es que el parámetro *a* de la función *f* sea 0; en este caso, la función *f* lanza (**throw**) una excepción de la clase *EValorNoValido* creando un objeto de esta clase. Para crear ese objeto se invoca al constructor *EValorNoValido* pasando como argumento, en este caso, la cadena “Error: valor cero”.

Especificación de excepciones

Si una función *f* lanza una excepción puede declararlo, para que los usuarios de la misma estén informados sobre las cosas anormales que puede hacer dicha función (algunos compiladores ignoran la especificación de excepciones).

```
void f(double) throw(EValorNoValido);

int main()
{
    double d = 0;
    cout << "dato: "; d = leerDouble();
    // ...
    try
    {
        f(d);
    }
}
```

```

    catch(EValorNoValido& e)
    {
        cout << e.what() << endl;
    }
    // ...
}

```

```

void f(double a) throw(EValorNoValido)
{
    if (a == 0)
        throw EValorNoValido("Error: valor cero");
    // ...
}

```

La especificación de excepción *throw(EValorNoValido)* indica que *f* puede lanzar la excepción *EValorNoValido* y las de sus derivadas, cuando existan. La ausencia de argumentos de tipo indica que la función no lanza excepciones, y la ausencia de la especificación de excepción indica que la función puede lanzar cualquier excepción. El siguiente ejemplo aclara lo expuesto:

```

void f();           // f puede lanzar cualquier excepción
void f() throw();  // f no lanza excepciones
void f() throw(X); // f sólo puede lanzar excepciones de tipo X
void f() throw(X, Y); // f sólo puede lanzar excepciones de tipo X
                    // y de tipo Y

```

Cuando se utilice una, debe aparecer tanto en la declaración como en la definición de la función, y afecta única y exclusivamente a la función que la declara, independientemente de que ésta pudiera llamar a otras funciones.

La ventaja de especificar las excepciones que una función puede lanzar es que como las declaraciones de las funciones pertenecen a una interfaz a la que los usuarios tendrán acceso, éstos saben qué excepciones deberán capturar.

Excepciones no esperadas

Ahora bien, ¿qué sucede si la función lanza una excepción imprevista, esto es, una excepción que no está incluida en la especificación de excepciones? En un caso como éste, se invocará a la función **std::unexpected** (éste es el comportamiento de C++ estándar; no obstante, algunos compiladores basados en el estándar no soportan esto). Estas llamadas son normalmente indeseables (excepto durante la depuración de un proyecto) porque esta función invoca a su vez a la función especificada por **set_unexpected**, que por omisión es **std::terminate** y que normalmente invoca a **abort**, interrumpiendo de forma brusca la ejecución de la aplicación. Si queremos evitar esto, tendremos que modificar el comportamiento de

unexpected. Una forma sencilla de hacer esto es añadiendo a la especificación de excepciones la excepción **bad_exception** de la biblioteca estándar de C++ y forzando a que **unexpected** lance esa excepción. De esta forma, cuando ocurra una excepción inesperada, se invocará a **unexpected** que lanzará la excepción **bad_exception** programada, que podremos capturar y manejar. Por ejemplo:

```
// ...
void f(double) throw(EValorNoValido, bad_exception);

int main()
{
    double d = 0;
    cout << "dato: "; d = leerDouble();
    // ...
    cout << d << endl;

    try
    {
        f(d);
    }
    catch(EValorNoValido& e)
    {
        cout << e.what() << endl;
    }
    catch(bad_exception& e)
    {
        cout << "error: " << e.what() << endl;
    }
    // ...
}

void excepcion_inesperada()
{
    throw std::bad_exception();
}

void f(double a) throw(EValorNoValido, bad_exception)
{
    set_unexpected(excepcion_inesperada);
    if (a == 0)
        throw EValorNoValido("Error: valor cero");
    // ...
    // Supongamos que en este punto ocurre una excepción inesperada
    throw "excepción inesperada";
    // ...
}
```

En este ejemplo se puede observar que la especificación de excepciones de *f* incluye la excepción **bad_exception**, entre otras. A su vez, *f* establece por medio

de `set_unexpected` que cuando sea invocada `unexpected` porque ocurrió una excepción inesperada, se ejecute la función `excepcion_inesperada` que lanzará la excepción `bad_exception` que podremos manejar análogamente a como lo hace la función `main`.

FLUJO DE EJECUCIÓN

La sentencia `throw EValorNoValido(...)` lanza la excepción `EValorNoValido`; esto es, crea un objeto de esta clase, que interrumpe el flujo de ejecución de la aplicación y vuelve por la pila de llamadas de funciones hasta encontrar una que sepa capturar la excepción (`catch`). En el camino de vuelta se llama a los destructores correspondientes a los objetos locales que van quedando atrás y que, lógicamente, fueron construidos desde que se inició el bloque `try`. El flujo de ejecución de la aplicación se transfiere entonces directamente a la función que capturó la excepción para que ejecute el manejador. Si el manejador, una vez ejecutado, permite que la aplicación continúe, el flujo de ejecución se transfiere a la primera línea ejecutable que haya a continuación del último manejador del bloque `try`. Para aclarar lo expuesto, analice y observe el resultado del ejemplo siguiente:

```
// excepciones.cpp - Tratamiento de excepciones
#include <string>
#include <iostream>
using namespace std;

class EValorNegativo
{
    string mensajeDeError;
public:
    EValorNegativo(string mensaje = "error")
    {
        mensajeDeError = mensaje;
    }
    string what() const { return mensajeDeError; }

    // ...
};

class CDemo
{
    int x;
public:
    CDemo()
    {
        x = 0;
        cout << "Se construye un objeto CDemo" << endl;
    }
};
```

```
~CDemo()
{
    cout << "Se destruye un objeto CDemo" << endl;
}

void AsignarValor(int v)
{
    if (v < 0)
    {
        cout<< "Ocurrió una excepción EValorNegativo\n";
        throw EValorNegativo("EValorNegativo");
    }
    else if (v == 0)
    {
        cout<< "Ocurrió una excepción; valor cero\n";
        throw "valor cero";
    }
    x = v;
}

// ...
};

void MiFuncion(int);

int main()
{
    int a = 0;
    cout << "a = "; cin >> a;
    cout << "Se ejecuta main\n";

    try
    {
        cout << "Se llama a MiFuncion desde el bloque try\n";
        MiFuncion(a);
        cout << "Finaliza el bloque try\n";
    }
    catch(EValorNegativo& e)
    {
        cout << "Se captura la excepción: ";
        cout << e.what() << endl;
    }
    catch(const char *s)
    {
        cout << s << endl;
    }
    cout << "Se reanuda la ejecución de main\n";

    // ...
}
```



```

void MiFuncion(int x)
{
    cout << "Se ejecuta MiFuncion\n";
    CDemo objD;
    objD.AsignarValor(x);
    // ...
    cout << "Continua la ejecución de MiFuncion\n";
    // ...
}

```

Si ejecuta este programa y fuerza a que la variable *x* tome un valor *negativo*, observará que se visualiza el resultado que se expone a continuación y que pone de manifiesto el proceso seguido cuando se lanza una excepción.

```

a = -1
Se ejecuta main
Se llama a MiFuncion desde el bloque try
Se ejecuta MiFuncion
Se construye un objeto CDemo
Ocurrió una excepción EValorNegativo
Se destruye un objeto CDemo
Se captura la excepción: EValorNegativo
Se reanuda la ejecución de main

```

Si un método, una función en general, lanza una excepción y en la vuelta por la pila de llamadas no se encuentra otra que la capture, el programa finalizará. En cambio, si se encuentra un manejador para esa excepción, se ejecuta. En el supuesto de que en la pila de llamadas quedaran otras funciones que pudieran capturarla, no serán tenidas en cuenta; esto es, sólo se tiene en cuenta el manejador de la función por la que haya pasado el flujo de control más recientemente.

A su vez, si la función contiene una lista de manejadores sólo se ejecutará la correspondiente a la excepción lanzada; esto es, el comportamiento es el mismo que el de una sentencia **switch**, pero con la diferencia de que los **case** necesitan sentencias **break** y los **catch** no.

Según hemos visto, una excepción se captura en un bloque **catch** que declare un argumento de su clase o clase base; pero como lo que se lanza es un objeto (**throw** especifica una llamada al constructor de la clase de excepción), si necesitáramos transmitir información adicional desde el punto de lanzamiento al manejador, lo podemos hacer dotando de parámetros al constructor.

Una excepción se considera manejada desde el momento en que se entra en su manejador, así que cualquier otra excepción lanzada desde el cuerpo de éste deberá ser capturada por alguna otra función cuya llamada se encuentre en la pila; si

la excepción no es capturada, la aplicación finaliza. Esto hace que el siguiente código no provoque un bucle infinito:

```
try
{
    MiFuncion();

    // ...
}
catch(EValorNegativo e)
{
    throw EValorNegativo();
    // ...
}
```

Cuando se trate de una excepción particular de una determinada clase podemos implementarla como una clase interna de ésta. Por ejemplo, suponiendo que la excepción *EValorNegativo* sólo se puede producir en el contexto de la clase *CDemo*, podríamos simplificar el ejemplo anterior de la forma siguiente:

```
// excepciones-internas.cpp - Tratamiento de excepciones
#include <string>
#include <iostream>
using namespace std;
```

```
class CDemo
{
    int x;
public:
    class EValorNegativo {}; // clase de excepción

    CDemo()
    {
        x = 0;
        cout << "Se construye un objeto CDemo" << endl;
    }

    ~CDemo()
    {
        cout << "Se destruye un objeto CDemo" << endl;
    }

    void AsignarValor(int v)
    {
        if (v < 0)
        {
            cout << "Ocurrió una excepción EValorNegativo\n";
            throw EValorNegativo();
        }
    }
}
```

```

        else if (v == 0)
        {
            cout<< "Ocurrió una excepción; valor cero\n";
            throw "valor cero";
        }
        x = v;
    }
    // ...
};

void MiFuncion(int);

int main()
{
    int a = 0;
    cout << "a = "; cin >> a;
    cout << "Se ejecuta main\n";
    try
    {
        cout << "Se llama a MiFuncion desde el bloque try\n";
        MiFuncion(a);
        cout << "Finaliza el bloque try\n";
    }
    catch(CDemo::EValorNegativo& e)
    {
        cout << "Se captura la excepción: ";
        cout << "EValorNegativo" << endl;
    }
    catch(const char *s)
    {
        cout << s << endl;
    }
    cout << "Se reanuda la ejecución de main\n";
    // ...

    system("pause");
}

void MiFuncion(int x)
{
    cout << "Se ejecuta MiFuncion\n";
    CDemo objD;
    objD.AsignarValor(x);
    // ...
    cout << "Continua la ejecución de MiFuncion\n";
    // ...
}

```

Obsérvese que, en este caso, la función **main** captura la excepción *EValorNegativo* de *CDemo*: *CDemo::EValorNegativo*.

También podría suceder que la propia función que lanza la excepción la capturara, como puede observar en el ejemplo siguiente:

```
void f(double a) throw(EValorNoValido, bad_exception)
{
    set_unexpected(excepcion_inesperada);
    try
    {
        if (a == 0)
            throw EValorNoValido("Error: valor cero");
        // ...
    }
    catch(EValorNoValido& e)
    {
        cout << e.what() << endl;
    }
    catch(bad_exception& e)
    {
        cout << "error: " << e.what() << endl;
    }
    // ...
}
```

Lo que sucede es que escribir una función que lance una o más excepciones y que ella misma las capture es anticiparnos a las necesidades que pueda tener el usuario de esa función (o de esa clase que proporciona ese método) en cuanto al tratamiento de la excepción se refiere.

Combinar ambas formas (declarar la excepción y además capturarla) no sirve de nada cara al usuario, porque si una función lanza una excepción y la captura, en el supuesto de que en la pila de llamadas quedaran otras que pudieran capturarla, no serán tenidas en cuenta; esto es, sólo se ejecuta el manejador de la función por la que haya pasado el flujo de control más recientemente.

CUÁNDO UTILIZAR EXCEPCIONES Y CUÁNDO NO

No todos los programas necesitan responder lanzando una excepción a cualquier situación anómala que se produzca. Por ejemplo, si partiendo de unos datos de entrada estamos haciendo una serie de cálculos más o menos complejos con la única finalidad de observar unos resultados, quizás la respuesta más adecuada a un error sea interrumpir sin más el programa, no antes de haber lanzado un mensaje apropiado y haber liberado los recursos adquiridos que aún no hayan sido liberados. Otro ejemplo, podemos utilizar la clase de excepción **IndiceDeLaMatrizFueraDeLimites** para manejar el error que se produce cuando se rebasan los límites de una matriz, pero es más fácil utilizar una sentencia **if** para prevenir que esto no suceda.

En cambio, si estamos construyendo una biblioteca, estaremos obligados a evitar todos los errores que se puedan producir cuando su código sea ejecutado por cualquier programa que la utilice.

Por último, no todas las excepciones tienen que servir para manipular errores. Puede también manejar excepciones que no sean errores.

ADQUISICIÓN DE RECURSOS

Generalmente, cuando una aplicación adquiere un recurso (asigna un bloque de memoria, abre un fichero, etc.) es vital liberar el recurso antes de que dicha aplicación finalice. Por ejemplo, suponiendo que la llamada a una función *AdquirirRecurso* se hace desde un bloque `try`,

```
void AdquirirRecurso()
{
    // Adquirir recurso
    // Operar con el recurso
    // Liberar el recurso
}
```

¿qué sucede si se produce un error al operar con ese recurso? Una excepción puede hacer que se salga de la función *AdquirirRecurso* sin liberar el recurso. Para evitar este problema, podemos utilizar un constructor para adquirir el recurso y un destructor para liberarlo. Esto es:

```
class CAdquirirRecurso
{
    // Constructor: adquiere el recurso
    // Destructor: libera el recurso
    // Otros métodos
}
```

Desde esta definición, podremos adquirir un recurso construyendo un objeto *CAdquirirRecurso*. Por ejemplo:

```
void AdquirirRecurso()
{
    CAdquirirRecurso r;
    // Operaciones con r
}
```

Según lo estudiado anteriormente, el recurso quedará ahora liberado independientemente de que se salga de la función *AdquirirRecurso* normalmente o por

lanzamiento de una excepción, ya que en ambos casos el destructor será invocado automáticamente.

Un recurso que adquirimos con frecuencia es memoria para almacenar un determinado objeto. Sirva como ejemplo la clase *CVector* mostrada a continuación. En ella se puede observar un método *iniciar* para poner a cero los elementos de la matriz:

```
// cvector.h - Declaración de la clase CVector
#ifndef !defined( _CVECTOR_H_ )
#define _CVECTOR_H_

class CVector
{
private:
    double *vector; // puntero al primer elemento de la matriz
    int nElementos; // número de elementos de la matriz
protected:
    double *asignarMem(int);
public:
    CVector(int ne = 10); // crea un CVector con ne elementos
    CVector(const CVector&); // crea un CVector desde otro
    CVector& operator=(const CVector&); // asignación
    ~CVector(); // destructor
    double& operator[](int i) const;
    int longitud() const;
    void iniciar();
};
#endif // _CVECTOR_H_
```

```
// cvector.cpp - Definición de la clase CVector
#include <iostream>
#include "cvector.h"
using namespace std;

// Crear una matriz con ne elementos
CVector::CVector(int ne)
{
    if (ne < 1) ne = 10;
    nElementos = ne;
    vector = asignarMem(nElementos);
}

// Constructor copia
CVector::CVector(const CVector &v)
{
    vector = 0;
    *this = v;
}
```

```

// Operador de asignación
CVector& CVector::operator=(const CVector &v)
{
    delete [] vector;
    nElementos = v.nElementos;
    vector = asignarMem(nElementos);
    // Copiar el objeto v
    for (int i = 0; i < nElementos; i++)
        vector[i] = v.vector[i];
    return *this;
}

// Otros métodos
CVector::~CVector() // destructor
{
    delete [] vector;
}

double& CVector::operator[](int i) const
{
    return vector[i];
}

int CVector::longitud() const { return nElementos; }

double *CVector::asignarMem(int nElems)
{
    try
    {
        double *p = new double[nElems];
        return p;
    }
    catch(bad_alloc e)
    {
        cout << "Insuficiente espacio de memoria\n";
        exit(-1);
    }
}

void CVector::iniciar()
{
    cout << "iniciando el vector...\n";
    fill(vector, vector + nElementos, 0);
    // Para simular que ocurre una excepción permita que se ejecute:
    // throw "excepción inesperada";
}

```

Observe el método *iniciar*. Una excepción inesperada durante su ejecución daría lugar a una terminación anómala del programa *test.cpp* que se muestra a

continuación y a que se generasen lagunas de memoria porque no se liberaría el espacio de memoria adquirido para el vector.

```
// test.cpp - Adquisición de recursos
#include <iostream>
#include "cvector.h"
using namespace std;

void f()
{
    int i = 0, n = 3;
    CVector *vector1;

    // Adquirir el recurso
    vector1 = new CVector(n);
    vector1->iniciar();

    // Operar con el recurso
    for (i = 0; i < vector1->longitud(); i++)
        (*vector1)[i] = i * 3;
    CVector vector2 = *vector1;
    for (i = 0; i < vector1->longitud(); i++)
        cout << vector2[i] << " ";
    cout << endl;

    // Liberar el recurso
    delete vector1;
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    {
        cout << "excepción inesperada\n";
        system("pause");
        return -1;
    }
    // Otras operaciones
}
```

Según lo expuesto, una forma segura de solucionar el problema planteado es utilizar un constructor para adquirir el recurso y un destructor para liberarlo. Para ello, construiremos una clase *Vector* cuyo constructor y destructor se encarguen de adquirir y liberar el recurso “espacio de memoria” necesario para la matriz. El constructor de la clase *Vector*, además de adquirir el recurso, iniciará la matriz

(método que puede fallar y lanzar una excepción); por lo tanto, eliminaremos el método *iniciar* de la clase *CVector* que escribimos anteriormente y lo incluiremos en la clase *Vector* que se muestra a continuación:

```
// vector.h - Declaración de la clase Vector
#ifndef _VECTOR_H_
#define _VECTOR_H_
#include "cvector.h"

class Vector
{
private:
    CVector *pvector;
public:
    Vector(int);
    ~Vector();
    Vector(const Vector& v);
    Vector& operator=(const Vector& v);
    double& operator[](int) const;
    int longitud() const;
    void iniciar();
};
#endif // _VECTOR_H_
```

```
// vector.cpp - Definición de la clase Vector
#include <iostream>
#include "vector.h"
using namespace std;

// Constructor
Vector::Vector(int n)
{
    pvector = new CVector(n);
}

// Destructor
Vector::~~Vector()
{
    delete pvector;
}

Vector::Vector(const Vector& v)
{
    pvector = 0;
    *this = v;
}

Vector& Vector::operator=(const Vector& v)
{
    pvector = new CVector(*v.pvector);
```

```
    return *this;
}

// Operador de indexación
double& Vector::operator[](int i) const
{
    return (*pvector)[i];
}

void Vector::iniciar()
{
    cout << "iniciando el vector...\n";
    for (int i = 0; i < pvector->longitud(); i++)
        (*pvector)[i] = 0;
    // Para simular que ocurre una excepción permita que se ejecute:
    throw "excepción inesperada";
}

int Vector::longitud() const
{
    return pvector->longitud();
}
```

Obsérvese que ahora el atributo *pvector* es un puntero a un objeto de la clase *CVector*. Esto quiere decir que cuando se cree un objeto de la clase *Vector*, primero hay que invocar al constructor de la clase *CVector* (véase el constructor *Vector*). De esta forma, cuando se inicie el objeto *Vector* (*vector1* en el ejemplo que mostramos a continuación) invocando a su método *iniciar*, ya ha finalizado la construcción del objeto *CVector* que ha adquirido el recurso. Por lo tanto, si *iniciar* lanza una excepción, se llamará al destructor del objeto de la clase *Vector*, liberándose automáticamente el recurso.

```
// test.cpp - Adquisición de recursos
#include <iostream>
#include "vector.h"
using namespace std;

void f()
{
    int i = 0, n = 3;

    // Adquirir el recurso
    Vector vector1(n);
    vector1.iniciar();

    // Operar con el recurso
    for (i = 0; i < vector1.longitud(); i++)
        vector1[i] = i * 3;
```

```

Vector vector2 = vector1;
for (i = 0; i < vector1.longitud(); i++)
    cout << vector2[i] << " ";
cout << endl;

// Liberar el recurso
// se invoca al destructor
}

int main()
{
    try
    {
        f();
    }
    catch(...)
    {
        cout << "excepción inesperada\n";
        return -1;
    }
    // Otras operaciones
}

```

Punteros inteligentes

La biblioteca estándar proporciona una plantilla denominada **auto_ptr** que soporta la adquisición de recursos. Básicamente un puntero automático es un objeto **auto_ptr** que almacena un puntero a otro objeto obtenido vía **new** y borra este objeto cuando él es destruido. Esta plantilla está declarada en el fichero `<memory>`. Veamos un ejemplo. Muchas veces escribimos código similar al siguiente:

```

void f(int n)
{
    Vector<double> *p = new Vector<double>(n);
    int lon = p->longitud();
    // Otro código que puede lanzar alguna excepción
    // ...
    delete p;
}

```

Si cuando se ejecute *f* no ocurre nada excepcional, todo habrá ido bien, pero si *f* lanza una excepción y no se ejecuta **delete**, entonces el objeto creado dinámicamente no será liberado y tendremos la clásica laguna de memoria. El problema planteado se podría evitar si el puntero fuera inteligente, esto es, que cuando fuera destruido liberara el objeto apuntado. Esto es justamente lo que hace un **auto_ptr**: crea dinámicamente un objeto de tipo *T* y almacena su dirección y, cuando se des-

truye, libera el espacio asignado al objeto referenciado. Según esto, el código anterior podría escribirse de forma segura así:

```
void f(int n)
{
    auto_ptr<Vector<double> > p(new Vector<double>(n));
    int lon = p->longitud(); // o bien, p.get()->longitud()
    // Otro código que puede lanzar alguna excepción
    // ...
}
```

Ahora no importa que la función f lance una excepción cuando se ejecute, porque cuando el flujo de ejecución salga fuera del ámbito de p , su destructor liberará la memoria asignada al objeto de tipo $Vector<double>$. El método **get** devuelve el puntero encapsulado por p que apunta al objeto.

Además de los constructores y el destructor, la plantilla **auto_ptr** proporciona los siguientes operadores y métodos:

Operador/método	Significado
*	($*a$). Devuelve una referencia ($T&$) al objeto apuntado.
->	($a->x$). Devuelve el puntero (T^*) al objeto.
=	($b = a$). Copia a en b . El objeto b pasa a ser el nuevo propietario del objeto apuntado y a pasa a no apuntar a nada, y si b ya es propietario de un objeto, ese objeto es liberado.
X *get() const	Igual que el operador $->$.
X *release()	Libera la propiedad del objeto apuntado por el auto_ptr y devuelve el puntero.
void reset($T^* p$)	Asigna p al puntero encapsulado por el objeto auto_ptr y libera el objeto actualmente referenciado.

A continuación realizamos algunos ejemplos que muestran cómo se utilizan estos operadores y métodos. El siguiente ejemplo define un tipo de estructuras t_tfno , declara un objeto pa de la clase $auto_ptr<Vector<t_tfno> >$, lo inicia con un puntero a un objeto de tipo $Vector<t_tfno>$, asigna datos a los elementos de la matriz encapsulada por el objeto y finalmente los muestra. Se puede observar que para acceder a uno de los elementos de la matriz necesitamos tener acceso al objeto vector, lo cual se logra a través del operador $*$ del objeto pa .

```
// test.cpp - Punteros inteligentes
#include <iostream>
#include <string>
#include <memory> // para auto_ptr
#include "vector.h" // plantilla Vector
#define LONGNOM 30
```

```

using namespace std;

typedef struct
{
    char nombre[LONGNOM];
    long tfno;
} t_tfno;

void leer(auto_ptr<Vector<t_tfno> >& pa)
{
    for (int i = 0; i < (*pa).longitud(); i++)
    {
        cout << "Nombre:   "; cin.getline((*pa)[i].nombre, LONGNOM);
        cout << "Teléfono: "; cin >> ((*pa)[i].tfno);
        cin.ignore();
    }
}

void mostrar(auto_ptr<Vector<t_tfno> >& pa)
{
    for (int i = 0; i < (*pa).longitud(); i++)
    {
        cout << "Nombre:   " << ((*pa)[i].nombre) << endl;
        cout << "Teléfono: " << ((*pa)[i].tfno) << endl;
    }
}

int main()
{
    auto_ptr<Vector<t_tfno> > pa;
    int n = 3;
    pa = auto_ptr<Vector<t_tfno> >(new Vector<t_tfno>(n));
    leer(pa);
    mostrar(pa);
}

```

Observe también que a la función *leer* se le pasa el objeto *pa* por referencia. ¿Se podría haber pasado por valor? Pues no. Para ver por qué, veamos qué ocurre cuando se copia un **auto_ptr** en otro. En este caso, el **auto_ptr** origen transfiere su propiedad (la del objeto referenciado) al **auto_ptr** destino, de forma que sólo éste posee el puntero a ese objeto, el cual será liberado cuando se destruya el **auto_ptr** destino. Si el **auto_ptr** destino ya posee un objeto, éste es liberado, mientras el **auto_ptr** origen es puesto a cero (el puntero que apuntaba al objeto que poseía pasa a valer 0). El siguiente ejemplo aclara lo expuesto:

```

auto_ptr<Vector<t_tfno> > pta1(new Vector<t_tfno>(n));
auto_ptr<Vector<t_tfno> > pta2;
leer(pta1);    // correcto.
pta2 = pta1;   // ahora pta2 posee el puntero y pta1 no.

```

```
mostrar(pta2); // correcto.
```

Cuando el flujo de ejecución sale fuera del ámbito de *pta1* y *pta2*, el destructor de **auto_ptr** borra el objeto apuntado por *pta2*, pero no hace nada sobre *pta1* porque tiene un puntero que vale 0.

Partiendo de este último ejemplo, mostramos a continuación otros ejemplos para ver cómo se utilizan el resto de los operadores y métodos de **auto_ptr**:

```
// Utilización de reset
Vector<t_tfno> *p = new Vector<t_tfno>(1);
cout << "Nombre:  "; cin.getline((*p)[0].nombre, LONGNOM);
cout << "Teléfono: "; cin >> (*p)[0].tfno;
pta2.reset(p);
p = 0; // ahora pta2 posee el objeto vector y p no
mostrar(pta2);
// Utilización de ->
long ne = pta2->longitud(); // devuelve el número de elementos
cout << "Número de elementos: " << ne << endl;
// Utilización de *
ne = (*pta2).longitud(); // devuelve el número de elementos
cout << "Número de elementos: " << ne << endl;
// Utilización de release
p = pta2.release(); // ahora p posee el objeto vector y pta2 no
cout << "Nombre:  " << (*p)[0].nombre << endl;
cout << "Teléfono: " << (*p)[0].tfno << endl;
delete p;
```

Algo que nunca debe hacerse es lo siguiente, aunque si lo intenta, lo más normal es que su compilador le avise de que no puede proceder así:

```
vector<auto_ptr<t_tfno> > lista;
// ...
sort(lista.begin(), lista.end());
```

No es seguro almacenar objetos **auto_ptr<T>** en contenedores estándar porque los algoritmos que trabajan sobre estos contenedores pueden realizar copias de unos objetos en otros, con lo que se pueden borrar objetos. Por ejemplo, en la operación de clasificación, dependiendo de cómo esté desarrollado el algoritmo y según lo expuesto anteriormente, es probable que algún objeto de los referenciados por los **auto_ptr** se borre. Claramente, los punteros inteligentes, en general, resuelven perfectamente el problema de las lagunas de memoria que pueden producirse cuando se trabaja con punteros.

EJERCICIOS RESUELTOS

1. Añadir a la aplicación realizada en el capítulo 9 sobre el mantenimiento de una lista de teléfonos el código necesario para que el método *registro* lance la excepción “índice fuera de límites” cuando sea preciso. Recuerde que el método *registro* devolvía el objeto *CPersona* que estaba en la posición *i* de la matriz *listaTelefonos* o un objeto con valores nulos si la posición especificada estaba fuera de límites. Implementando el código solicitado evitaremos que el método tenga que devolver, en caso de error, un objeto con valores nulos que es costoso de manipular. Este método una vez modificado puede ser así:

```
CPersona CListaTfnos::registro(unsigned int i)
{
    if (i >= 0 && i < listaTelefonos.size())
        return listaTelefonos[i];
    else
        throw "Índice fuera de límites";
}
```

Como ejemplo de tratamiento de este error, vamos a añadir a la función *buscar* de la aplicación “lista de teléfonos” realizada en el capítulo 9 un manejador para esta excepción. Cargue esta aplicación, visualice el fichero *test.cpp* y diríjase a la función *buscar*. Después modifíquela como se indica a continuación:

```
void buscar(CListaTfnos& listatfnos, bool buscar_siguiente)
{
    static int pos = -1;
    static string cadenabuscar;

    if (!buscar_siguiente)
    {
        cout << "conjunto de caracteres a buscar: ";
        getline(cin, cadenabuscar);
        // Buscar a partir del principio
        pos = listatfnos.buscar(cadenabuscar, 0);
    }
    else
        // Buscar el siguiente a partir del último encontrado
        pos = listatfnos.buscar(cadenabuscar, pos+1);

    try
    {
        cout << listatfnos.registro(pos).obtenerNombre() << endl;
        cout << listatfnos.registro(pos).obtenerDireccion() << endl;
        cout << listatfnos.registro(pos).obtenerTelefono() << endl;
    }
    catch(const char *msj)
    {
        if (listatfnos.longitud() != 0)
```

```

        cout << "búsqueda fallida o " << msj << endl;
    else
        cout << "lista vacía\n";
    }
}

```

2. Al principio de este capítulo implementamos una función como la siguiente:

```

double leerDouble()
{
    cin.exceptions(ios::failbit | ios::badbit);
    double dato = 0.0;
    try
    {
        cin >> dato;
    }
    catch(ios_base::failure& e)
    {
        cout << e.what() << ": dato no válido\n";
        cin.clear();
        cin.ignore(numeric_limits<int>::max(), '\n');
    }
    return dato; // devolver el dato tecleado
}

```

Esta función devuelve el valor de tipo **double** obtenido de la entrada estándar. Pero, ¿qué ocurre si la cadena de caracteres tecleada no se corresponde con un **double**? Pues que al ejecutarse el método **operator>>** de **cin**, se lanza una excepción **ios_base::failure** que es capturada por el manejador para desactivar los indicadores de error y limpiar el búfer de la entrada estándar. Finalmente, la función devuelve el valor actual de *dato* (0.0 en caso de error).

Una alternativa al manejador anterior podría ser otro que ante un dato no válido (por ejemplo: xxx) solicitara teclear un dato correcto. También conviene que nuestra función informe de cómo transcurrió su ejecución devolviendo, por ejemplo, un valor **true** (ejecución satisfactoria) o **false**. Esto obliga a pasar a la función la variable a leer por referencia. Así mismo, la función devolverá **false** cuando se teclee el carácter fin de fichero (*Ctrl+Z*). De esta forma, podremos utilizar *Ctrl+Z* como marca para finalizar una entrada masiva de datos. Según esto, podemos reescribir el método *leerDouble* así:

```

bool leerDouble(double& dato)
{
    cin.exceptions(ios::failbit | ios::badbit);
    try
    {
        cin >> dato;
        // Eliminar caracteres sobrantes. Por ejemplo <Entrar>.
    }
}

```



```

        cin.ignore(numeric_limits<int>::max(), '\n');
        return true;
    }
    catch(ios_base::failure& e)
    {
        if (cin.eof())
        {
            cin.clear();
            return false; // se pulsó Ctrl+Z
        }
        else
        {
            cout << e.what() << ": dato no válido\n";
            cin.clear();
            cin.ignore(numeric_limits<int>::max(), '\n');
            return leerDouble(dato);
        }
    }
}

```

El método **max** de la clase **numeric_limits<T>** devuelve el valor máximo para el tipo *T*.

Se puede observar que ante una entrada no válida, el manejador llama recursivamente a la función, excepto si se pulsó *Ctrl+Z*.

Un ejemplo de utilización de esta función puede ser el siguiente:

```

int main()
{
    double d = 0;
    while(leerDouble(d))
        cout << d << endl;
}

```

Como ejercicio, se trata ahora de escribir una plantilla de función *leerDato* con el siguiente prototipo:

```

template<class T> bool leerDato(T& dato);

```

Esta plantilla de función permitirá leer cualquier tipo de datos que pueda leer el método **operator>>** de **cin**. Para leer objetos de tipo **string** utilizaremos el método **getline** de esta clase, ya que este método no interpreta el espacio en blanco como separador como lo hace **operator>>** de **cin**, sino que lee caracteres hasta encontrar el carácter ‘\n’ incluido éste. Esto supone particularizar la plantilla anterior para el tipo **string**. De acuerdo con lo expuesto, una posible solución puede ser la siguiente:

```
// leerdatos.h - Funciones para leer datos de forma segura
#ifndef _LEERDATOS_H_
#define _LEERDATOS_H_

#include <iostream>
#include <limits>
#include <string>

template<class T> bool leerDato(T& dato)
{
    std::cin.exceptions(std::ios::failbit | std::ios::badbit);
    try
    {
        std::cin >> dato;
        // Eliminar caracteres sobrantes. Por ejemplo <Entrar>.
        std::cin.ignore(std::numeric_limits<int>::max(), '\n');
        return true;
    }
    catch(std::ios_base::failure& e)
    {
        if (std::cin.eof())
        {
            std::cin.clear();
            return false; // se pulsó Ctrl+Z
        }
        else
        {
            std::cout << e.what() << ": dato no válido\n";
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<int>::max(), '\n');
            return leerDato(dato);
        }
    }
}

bool leerDato(std::string& dato);

#endif // _LEERDATOS_H_
```

```
// leerdatos.cpp - Definiciones de funciones
#include "leerdatos.h"
using namespace std;

bool leerDato(string& dato)
{
    bool v = true;

    cin.exceptions(ios::failbit | ios::badbit);

    try
```

```

{
    getline(cin, dato); // leer una variable de tipo string
    v = true;
}
catch(ios_base::failure& e)
{
    if (cin.eof())
    {
        cin.clear();
        v = false; // se pulsó Ctrl+Z
    }
}
return v;
}

```

El programa *test.cpp* que se expone a continuación, pone a prueba el código anterior. Este programa presentará un menú como el siguiente:

-
1. Leer un char
 2. Leer un short
 3. Leer un int
 4. Leer un float
 5. Leer un double
 6. Leer un string
 7. Salir
-

Opción (1 - 7):

Para ello, puede añadir la siguiente función al fichero *leerdatos.cpp* anterior y su prototipo al fichero *leerdatos.h*:

```

int menu(char *opciones[], int numOpciones)
{
    int i;
    int opcion = 0;

    cout << "\n\n_____ \n\n";
    for (i = 1; i <= numOpciones; ++i)
        cout << "    " << i << ". " << opciones[i-1] << endl;
    cout << "_____ \n";
    do
    {
        cout << "\nOpción (1 - " << numOpciones << "): ";
        leerDato(opcion);
    }
    while (opcion < 1 || opcion > numOpciones);
}

```

```
    return opcion;
}
```

Finalmente, escribimos el programa *test.cpp* así:

```
// test.cpp - Funciones para leer datos de forma segura
#include <iostream>
#include <string>
#include "leerdatos.h"
using namespace std;

int main()
{
    char  cDato;
    short hDato;
    int   iDato;
    float fDato;
    double dDato;
    string sDato;
    bool salir = false;

    // Opciones del menú
    static char *opciones[] =
    {
        "Leer un char",
        "Leer un short",
        "Leer un int",
        "Leer un float",
        "Leer un double",
        "Leer un string",
        "Salir"
    };

    int nOpciones = sizeof(opciones)/sizeof(char*);
    do
    {
        switch (menu(opciones, nOpciones))
        {
            case 1:
                cout << "char: "; leerDato(cDato);
                cout << "Dato leído: " << cDato << endl;
                break;
            case 2:
                cout << "short: "; leerDato(hDato);
                cout << "Dato leído: " << hDato << endl;
                break;
            case 3:
                cout << "int: "; leerDato(iDato);
                cout << "Dato leído: " << iDato << endl;
                break;
        }
    } while (!salir);
}
```

```

    case 4:
        cout << "float: "; leerDato(fDato);
        cout << "Dato leído: " << fDato << endl;
        break;
    case 5:
        cout << "double: "; leerDato(dDato);
        cout << "Dato leído: " << dDato << endl;
        break;
    case 6:
        cout << "string: "; leerDato(sDato);
        cout << "Dato leído: " << sDato << endl;
        break;
    case 7:
        salir = true;
        break;
}
}
while(!salir);
}

```

EJERCICIOS PROPUESTOS

1. Partiendo de las clases *CEstudios*, *CAumno*, *CAsignatura* y sus derivadas, *CConvocatoria* y *CFecha* construidas en el capítulo 11, modifique el método *CAumno::asignarDNI* para que en vez de visualizar el mensaje, lance una excepción que permita visualizar ese mensaje.

Observe la opción *Matricular* de la aplicación realizada con las clases anteriores (fichero *test.cpp*). Atrape la excepción anterior de modo que no se añada el nuevo alumno y se visualice el mensaje generado al lanzar la excepción.

Modifique el método *CEstudios::alumno* para que en vez de visualizar el mensaje, lance una excepción que permita visualizar ese mensaje.

Atrape la excepción lanzada por *CEstudios::alumno*.

2. La clase *CCuenta* que implementamos en el capítulo 11 tiene un método *reintegro* que muestra un mensaje “Error: no dispone de saldo” cuando se intenta retirar una cantidad y no hay suficiente saldo. Modifique esta clase para que el método *reintegro* lance una excepción del tipo *ESaldoInsuficiente*.

La clase *ESaldoInsuficiente* tendrá dos atributos: uno de la clase *CCuenta* para hacer referencia a la cuenta que causó el problema y otro de tipo **double** para almacenar la cantidad solicitada. Así mismo tendrá un constructor y el método *mensaje*. El constructor *ESaldoInsuficiente* tendrá dos parámetros que harán referencia a la cuenta causante del problema y a la cantidad solicitada. El método

mensaje no tiene argumentos, generará un mensaje de error basado en la información almacenada en los atributos y devolverá un objeto **string** con ese mensaje.

Atrape la excepción *ESaldoInsuficiente* en la parte de la aplicación que considere más adecuada.

Cuando haya finalizado pruebe la jerarquía de la clase *CCuenta* junto con la clase *CBanco* que también implementamos en ese capítulo.

CAPÍTULO 14

© F.J.Ceballos/RA-MA

FLUJOS

Todos los programas realizados hasta ahora obtenían los datos necesarios para su ejecución de la entrada estándar y visualizaban los resultados en la salida estándar. Por otra parte, una aplicación podrá retener los datos que manipula en su espacio de memoria, sólo mientras esté en ejecución; es decir, cualquier dato introducido se perderá cuando la aplicación finalice.

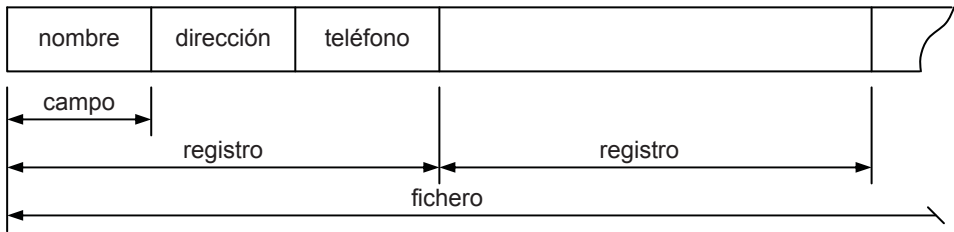
Por ejemplo, si hemos realizado un programa con la intención de construir una agenda, lo ejecutamos y almacenamos los datos *nombre*, *apellidos* y *teléfono* de cada uno de los componentes de la agenda en una matriz, los datos estarán disponibles mientras el programa esté en ejecución. Si finalizamos la ejecución del programa y lo ejecutamos de nuevo, tendremos que volver a introducir de nuevo todos los datos.

La solución para hacer que los datos persistan de una ejecución a otra es almacenarlos en un fichero en el disco en vez de en una matriz en memoria. Entonces, cada vez que se ejecute la aplicación que trabaja con esos datos, podrá leer del fichero los que necesite y manipularlos. Nosotros procedemos de forma análoga en muchos aspectos de la vida ordinaria; almacenamos los datos en fichas y guardamos el conjunto de fichas en lo que generalmente denominamos fichero o archivo.



Desde el punto de vista informático, un fichero o archivo es una colección de información que almacenamos en un soporte físico (por ejemplo, un disco magnético o un CD) para poderla manipular en cualquier momento. Esta información se almacena como un conjunto de registros, conteniendo todos ellos, generalmente, los mismos campos. Cada campo almacena un dato de un tipo predefinido o definido por el usuario. El registro más simple estaría formado por un carácter.

Por ejemplo, si quisiéramos almacenar en un fichero los datos relativos a la agenda de teléfonos a la que nos hemos referido anteriormente, podríamos diseñar cada registro con los campos *nombre*, *dirección* y *teléfono*. Según esto y desde un punto de vista gráfico, puede imaginarse la estructura del fichero así:



Cada campo almacenará el dato correspondiente. El conjunto de campos descritos forma lo que hemos denominado registro, y el conjunto de todos los registros forman un fichero que almacenaremos, por ejemplo, en el disco bajo un nombre.

Por lo tanto, para manipular un fichero que identificamos por un nombre, son tres las operaciones que tenemos que realizar: abrir el fichero, escribir o leer registros del fichero y cerrar el fichero. En la vida ordinaria hacemos lo mismo: abrimos el cajón que contiene las fichas (fichero), cogemos una ficha (registro) para leer datos o escribir datos y, finalizado el trabajo con la ficha, la dejamos en su sitio y cerramos el cajón de fichas (fichero).

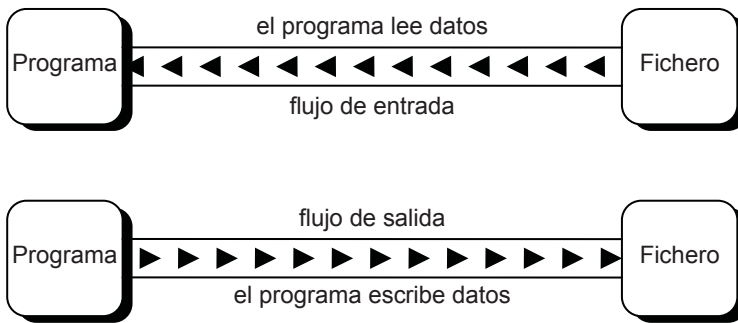
En programación orientada a objetos, hablaremos de objetos más que de registros y de sus atributos más que de campos.

Podemos agrupar los ficheros en dos tipos: ficheros de la aplicación (son los ficheros *.h*, *.cpp*, *.exe*, etc. que forman la aplicación) y ficheros de datos (son los que proveen de datos a la aplicación). A su vez, C++ ofrece dos tipos diferentes de acceso a los ficheros de datos: secuencial y aleatorio.

Para dar soporte al trabajo con ficheros, la biblioteca de C++ proporciona varias clases de entrada/salida (E/S) que permiten leer y escribir datos a, y desde, ficheros y dispositivos (en el capítulo 2 trabajamos con algunas de ellas).

VISIÓN GENERAL DE LOS FLUJOS DE E/S

Sabemos que la comunicación entre un programa y el origen o el destino de cierta información se realiza mediante un *flujo* de información (en inglés *stream*) que no es más que un objeto que hace de intermediario entre el programa y el origen o el destino de la información. De esta forma, el programa leerá o escribirá en el *flujo* sin importarle desde dónde viene la información o adónde va.

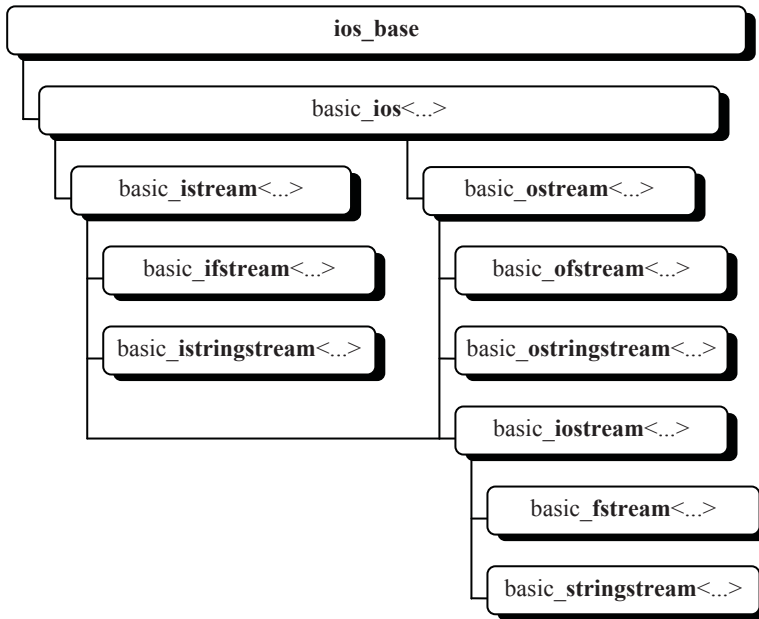


Este nivel de abstracción hace que un programa no tenga que saber nada del dispositivo, lo que se traduce en una facilidad más a la hora de escribir programas, ya que los algoritmos para leer y escribir datos serán siempre más o menos los mismos:

Leer	Escribir
<i>Abrir un flujo desde un fichero</i>	<i>Abrir un flujo hacia un fichero</i>
<i>Mientras haya información</i>	<i>Mientras haya información</i>
<i>Leer información</i>	<i>Escribir información</i>
<i>Cerrar el flujo</i>	<i>Cerrar el flujo</i>

La biblioteca estándar de C++ define, en su espacio de nombres **std**, una colección de clases que soportan estos algoritmos para leer y escribir. Por ejemplo, la clase **fstream**, subclase de **iostream**, permite escribir o leer datos de un fichero; análogamente, las clases **ifstream** y **ofstream**, subclases de **istream** y **ostream**, respectivamente, permiten definir flujos de entrada y de salida vinculados con ficheros; y las clases **istringstream**, **ostringstream** y **stringstream** permiten definir flujos, de entrada, de salida y de entrada-salida, respectivamente, vinculados con cadenas de caracteres.

La figura siguiente muestra las clases descritas y su posición en la jerarquía de clases definida en la biblioteca estándar de C++. Todas ellas serán estudiadas a continuación, excepto las que ya fueron estudiadas en el capítulo 2:



Exactamente, la clase **ios** (identificador que hemos escrito en **negrita**) se obtiene a partir la plantilla **basic_ios<...>** particularizada para datos de tipo **char**:

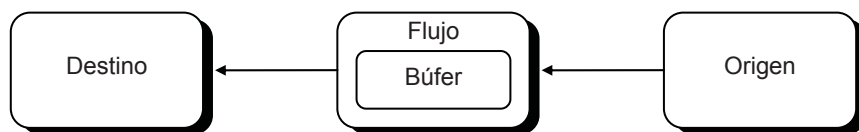
```
typedef basic_ios<char>      ios;
```

Análogamente, las clases **istream**, **ostream** e **iostream** para la E/S estándar se obtienen, respectivamente, a partir de las plantillas de clase **basic_istream**, **basic_ostream** y **basic_iostream** según muestra la figura anterior; las clases **ifstream**, **ofstream** y **fstream** para flujos de fichero se obtienen, respectivamente, a partir de las plantillas de clase **basic_ifstream**, **basic_ofstream** y **basic_fstream**; y las clases **istringstream**, **ostringstream** y **stringstream** para flujos de cadena se obtienen, respectivamente, a partir de las plantillas de clase **basic_istreamream**, **basic_ostringstream** y **basic_stringstream**.

BÚFERES

Generalmente, un flujo de salida coloca los caracteres en un búfer, y éste los almacena, como si de una matriz se tratara, hasta que un desbordamiento le fuerza a escribirlos en el destino real. Análogamente, un flujo de entrada toma caracteres de un búfer mientras los haya, hasta que la falta de ellos le fuerza a leer más del origen real. Dicho búfer es un objeto de la clase **streambuf**. Por lo tanto, esta clase aporta una característica muy interesante de la que se benefician todas las clases de E/S: una memoria intermedia para lecturas y escrituras futuras. Por

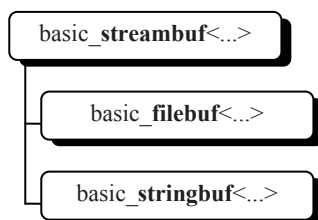
ejemplo, más adelante veremos que tanto un **ofstream** como un **ostream** se crean iniciando un **ostream** con un **streambuf** apropiado. Para entender esto observe la figura siguiente:



Según el esquema anterior, cuando una aplicación ejecute una sentencia de entrada (que solicite datos) los datos obtenidos del origen pueden ser depositados en el búfer en bloques más grandes que los que realmente está leyendo la aplicación (por ejemplo, cuando se leen datos de un disco la cantidad mínima de información transferida es un bloque equivalente a una unidad de asignación). Esto aumenta la velocidad de ejecución porque la siguiente vez que la aplicación necesite más datos no tendrá que esperar por ellos porque ya los tendrá en el búfer. Por otra parte, cuando se trate de una operación de salida, los datos no serán enviados al destino hasta que no se llene el búfer (o hasta que se fuerce el vaciado del mismo implícita o explícitamente), lo que reduce el número de accesos al dispositivo físico vinculado que siempre resulta mucho más lento que los accesos a memoria, aumentando por consiguiente la velocidad de ejecución.

Cuando el origen es el teclado y el destino el programa, el esquema es el mismo. Esto permite introducir los datos por anticipado para una aplicación en ejecución de la que se sabe que más adelante va a solicitarlos a través del teclado.

La figura siguiente muestra la clase **streambuf** y sus derivadas, todas ellas pertenecientes a la biblioteca estándar de C++:



Exactamente, la clase **streambuf** (identificador que hemos escrito en **negrita**) se obtiene a partir de la plantilla **basic_streambuf<...>** particularizada para datos de tipo **char**:

```
typedef basic_streambuf<char>          streambuf;
```

Análogamente, las clases **filebuf** y **stringbuf** se obtienen, respectivamente, a partir de las plantillas de clase **basic_filebuf**, y **basic_stringbuf** según muestra la figura anterior.

VISIÓN GENERAL DE UN FICHERO

Un fichero, independientemente de su tipo, es una secuencia de bytes almacenada en binario en un dispositivo de almacenamiento. Por ejemplo, si abrimos el código fuente de un supuesto programa *holamundo.cpp* con un editor de texto, se mostrarán cada una de las líneas que lo forman así:

```
/* holamundo.cpp */
#include <iostream>
using namespace std;

int main()
{
    cout << "¡Hola mundo!\n";
    system("pause");
    return 0;
}
```

Ahora bien, si lo abrimos con otro tipo de editor capaz de mostrarlo byte a byte en hexadecimal se mostraría lo siguiente:

```
00000000 2f 2a 20 68 6f 6c 61 6d 75 6e 64 6f 2e 63 70 70 /* holamundo.cpp
00000010 20 2a 2f 0d 0a 23 69 6e 63 6c 75 64 65 20 3c 69 */./.#include <i
00000020 6f 73 74 72 65 61 6d 3e 0d 0a 75 73 69 6e 67 20 ostream>..using
00000030 6e 61 6d 65 73 70 61 63 65 20 73 74 64 3b 0d 0a namespace std;..
00000040 0d 0a 69 6e 74 20 6d 61 69 6e 28 29 0d 0a 7b 0d ..int main()..{.
00000050 0a 20 20 63 6f 75 74 20 3c 3c 20 22 a1 48 6f 6c . cout << ".Hol
00000060 61 20 6d 75 6e 64 6f 21 5c 6e 22 3b 0d 0a 20 20 a mundo!\n";..
00000070 73 79 73 74 65 6d 28 22 70 61 75 73 65 22 29 3b system("pause");
00000080 0d 0a 20 20 72 65 74 75 72 6e 20 30 3b 0d 0a .. return 0;..}
00000090 0d 0a ..
```

Según podemos observar, el editor utilizado muestra el contenido del fichero en líneas de 16 bytes. En la columna de la izquierda se indica la posición del primer byte de cada fila, en la central aparecen los 16 bytes en hexadecimal y a la derecha aparecen los caracteres correspondientes a estos bytes. Sólo se muestran los caracteres imprimibles de los 128 primeros caracteres ASCII; el resto aparecen representados por un punto. El código ASCII coincide con los códigos ANSI y UNICODE sólo en los 128 primeros caracteres; en cambio, ANSI y UNICODE coinciden en los caracteres 0 a 255.

Por ejemplo, en la penúltima línea los bytes `72 65 74 75 72 6e` son los caracteres pertenecientes a la palabra clave **return**. A continuación aparecen `20 30 3b` que corresponden al espacio en blanco, el 0 y el punto y coma, y luego `0d 0a`. Los bytes `0d 0a` son el salto al principio de la línea siguiente, que en Windows se representa con dos caracteres. El `0d` es el ASCII *CR* (*Carriage Return*: retorno de carro) y el `0a` es el ASCII *LF* (*Line Feed*: avance de línea).

A pesar de que toda la información del fichero está escrita en 0 y 1 (en bits – en binario) cada byte (cada 8 bits) del fichero se corresponde con un carácter de la tabla de códigos de caracteres utilizada (ASCII, ANSI, UNICODE, etc.); por eso, estos ficheros son denominados *ficheros de texto*. Cuando no existe esta correspondencia hablamos de *ficheros binarios* sin más.

Las aplicaciones Windows, casi en su totalidad, utilizan el código de caracteres ANSI. Esto significa que si utilizamos una aplicación como el bloc de notas para escribir en un fichero el carácter ‘á’, en dicho fichero se almacenará el byte ‘e1’ (código 225). Si ahora, utilizando esa misma aplicación u otra, mostramos el contenido de ese fichero, se visualizará el carácter de código ‘e1’ que será ‘á’ si la aplicación trabaja con ANSI (caso del bloc de notas, *WordPad*, *Word*, *Visual C++*, etc.) o ‘β’ si utiliza el código ASCII (caso de una consola de Windows). Esto es, el carácter correspondiente a un determinado código depende de la tabla de códigos utilizada por la aplicación. Algunos ejemplos son:

Código Hex.	ASCII	ANSI	UNICODE
61	a	a	a
e1	β	á	á
f1	±	ñ	ñ

En Linux, actualmente, se usa UTF-8 y poco a poco va tomando presencia el UNICODE (código de 16 bits por carácter). Además, en Linux, y en UNIX en general, el carácter ‘n’ empleado por C para situar el punto de inserción al principio de la línea siguiente se codifica con un sólo carácter: el `0a` (*LF*). Según esto, para que los programas escritos en C se puedan utilizar en Linux y en Windows (recuerde que el lenguaje C originalmente se diseñó justamente para escribir UNIX en este lenguaje en vistas a su transportabilidad a otras máquinas), los compiladores de C para Windows han sido escritos para que traduzcan el carácter ‘n’ en la secuencia `0d 0a` al escribir texto en un dispositivo y viceversa, de `0d 0a` a sólo `0a`, al leer texto de un dispositivo.

Para entender lo expuesto vamos a realizar un pequeño ejemplo. Como veremos un poco más adelante, para escribir texto en un fichero se puede utilizar el operador << con el flujo que define el fichero en el que se quiere escribir. Previamente hay que abrir el fichero y al final hay que cerrarlo.

```

// crlf-t.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    ofstream ofs; // flujo
    // Abrir el fichero
    ofs.open("mifichero-t.txt", ios::out); // fichero de texto
    // Escribir en el fichero
    ofs << "¡Hola mundo!\n";
    ofs << 9 << " * " << 256 << " = " << 9*256 << "\n";
    // Cerrar el fichero
    ofs.close();
    return 0;
}

```

Cuando ejecute este programa se creará el fichero *mifichero-t.txt* con el contenido “¡Hola mundo!\n” más los números 9, 256 y 2304. Muestre el contenido de este fichero en hexadecimal:

```

00000000 a1 48 6f 6c 61 20 6d 75 6e 64 6f 21 0d 0a 39 20 .Hola mundo!..9
00000010 2a 20 32 35 36 20 3d 20 32 33 30 34 0d 0a * 256 = 2304..

```

Observe que el texto se ha escrito en ASCII, los caracteres ‘\n’ se han codificado como *0d 0a* y los números se han codificado también en ASCII, empleando un byte para cada dígito.

Modifique el programa añadiendo *binary* al modo en el que se abrirá el fichero (*binary* indica que se utilizará un fichero binario en lugar de un fichero de texto) y cambie el nombre de éste para que ahora se llame *mifichero-b.txt*:

```
ofs.open("mifichero-b.txt",ios::out|ios::binary);// fichero binario
```

Ejecute este programa. Se creará el fichero *mifichero-b.txt*. Muestre el contenido de este fichero en hexadecimal:

```

00000000 a1 48 6f 6c 61 20 6d 75 6e 64 6f 21 0a 39 20 2a .Hola mundo!.9 *
00000010 20 32 35 36 20 3d 20 32 33 30 34 0a 256 = 2304.

```

Comparando este resultado con el anterior vemos que ahora los caracteres ‘\n’ no se han traducido a *0d 0a*, si no que se han dejado como en UNIX: *0a*. El resto del contenido no ha cambiado; esto es, el texto y los números se siguen representando en ASCII.

Un ejemplo más, pero ahora utilizando la función **write**, que estudiaremos un poco más adelante, en lugar del operador <<:

```
// binario-b.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    int n = 0;
    char *s = "¡Hola mundo!\n";
    ofstream ofs; // flujo
    // Abrir el fichero
    ofs.open("mifichero-b.bin", ios::out|ios::binary);// fi. binario
    // Escribir en el fichero
    ofs.write(s, strlen(s));
    n = 9;          ofs.write((char *)&n, sizeof(int));
    n = 256;       ofs.write((char *)&n, sizeof(int));
    n = 17432583; ofs.write((char *)&n, sizeof(int));
    // Cerrar el fichero
    ofs.close();
    return 0;
}
```

Cuando ejecute este programa se creará el fichero *mifichero-b.bin* con el contenido “¡Hola mundo!\n” más los números 9, 256 y 17432583 (0x010a0007). Muestre el contenido de este fichero en hexadecimal:

```
00000000 a1 48 6f 6c 61 20 6d 75 6e 64 6f 21 0a 09 00 00 .Hola mundo!....
00000010 00 00 01 00 00 07 00 0a 01 .....
```

Observe que el texto se ha escrito en ASCII, el carácter ‘\n’ se ha codificado como *0a* (no hay conversión) porque se trata de un fichero binario y los números se han codificado en binario: cuatro bytes por cada **int** escritos de menor a mayor peso. Fíjese también que el último número tiene un byte que es *0a* que por formar parte de un número de cuatro bytes no tiene ningún significado especial (cuando se lea el número se leerán los cuatro bytes). De aquí se deduce que los números pueden escribirse en ASCII (<<) o en binario (**write**).

Si ahora intenta abrir el fichero *mifichero-b.bin* con un editor de texto, por ejemplo con el bloc de notas, observará que sólo es legible la información escrita en ASCII: el texto.

Modifique el programa quitando *binary* al modo en el que se abrirá el fichero (cuando se omite *binary* el fichero pasa a ser interpretado automáticamente de tex-

to), cambie el nombre al fichero para que ahora se llame *mifichero-t.bin* y cambie el tercer número por este otro: 17435911 (0x010a0d07):

```
ofs.open("mifichero-t.bin", ios::out); // fichero de texto
```

Ejecute este programa. Se creará el fichero *mifichero-t.bin*. Muestre el contenido de este fichero en hexadecimal:

```
00000000 a1 48 6f 6c 61 20 6d 75 6e 64 6f 21 0d 0a 09 00 .Hola mundo!....
00000010 00 00 00 01 00 00 07 00 0d 0a 01 .....

```

Comparando este resultado con el anterior vemos que ahora el carácter ‘\n’ se ha traducido a *0d 0a* porque se trata de un fichero de texto y, por lo tanto, el byte *0a* del tercer número también se ha traducido a *0d 0a* (por eso el número tiene cinco bytes en lugar de cuatro), para que en el proceso de lectura (abriendo el fichero para leer como fichero de texto), estos bytes vuelvan a ser traducidos en *0a* y el número quede inalterado. El resto del contenido no ha cambiado.

DESCRIPCIÓN DE LOS BÚFERES Y FLUJOS

Una vez descritas las jerarquías de clases que C++ proporciona para realizar la E/S y los mecanismos en los que ésta se fundamenta, es el momento de estudiar la funcionalidad que aporta cada una de las clases de esas jerarquías.

Clase **streambuf**

La clase **streambuf** es una clase base abstracta, de la cual se derivan las clases:

- | | |
|------------------|---|
| filebuf | Esta clase provee la funcionalidad correspondiente para crear un búfer y vincularlo a un fichero. |
| stringbuf | Esta clase provee la funcionalidad correspondiente para crear un búfer y vincularlo a una cadena de caracteres. |

Un objeto **streambuf** mantiene un área fija de memoria (búfer) que puede ser dividida dinámicamente en un *área de lectura* y en un *área de escritura*. Estas dos áreas pueden o no solaparse. La definición de esta clase incluye un *puntero de lectura*, que indica la posición del área de lectura a partir de la cual se realizará la siguiente operación de entrada, y un *puntero de escritura*, que indica la posición del área de escritura a partir de la cual se realizará la siguiente operación de salida. Dicha definición puede verla en el fichero *streambuf*. En ella puede observar los punteros necesarios para mantener el búfer. Hay también definida una serie de métodos (constructor, destructor virtual, etc.) que generalmente nosotros no utilizaremos directamente. Solamente, para algún caso particular, puede que sea nece-

sario redefinir los métodos **overflow** y **underflow**. El método **overflow** es llamado por los métodos **sputc** y **sputn** cuando el área de escritura está llena, y el método **underflow** es llamado por los métodos **sgetc** y **sgetn** cuando el área de lectura está vacía.

Clase **filebuf**

La clase **filebuf** es una clase derivada de la clase **streambuf** especializada en proveer búferes para gestionar la E/S sobre ficheros. Para un objeto **filebuf**, el área de lectura y el área de escritura son siempre la misma. Por lo tanto, los punteros de lectura y de escritura actúan como si fueran uno (cuando uno se mueve, también lo hace el otro). Para utilizar esta clase debe incluir en su código fuente la directriz:

```
#include <fstream>
```

La funcionalidad de esta clase está soportada por los siguientes métodos, entre otros (en la ayuda proporcionada en el CD puede ver todo acerca de ella):

```
filebuf();
~filebuf();
```

Estos dos métodos son el constructor y el destructor. El constructor crea un objeto **filebuf** sin conectarlo a un fichero (no abre un fichero), y el destructor cierra el fichero vinculado con el objeto **filebuf** y destruye este objeto.

```
filebuf *open(const char *nombre_fichero, ios_base::openmode modo);
```

Este método abre el fichero especificado por *nombre_fichero* y lo conecta con el objeto **filebuf** que recibe el mensaje **open**. Si el fichero ya está abierto (el método *is_open()* devuelve **true**) o si ocurre un error, el método **open** devuelve el valor 0; en otro caso, devuelve la dirección del objeto **filebuf**.

El parámetro *modo* determina qué tipo de operaciones pueden realizarse sobre el fichero. Se trata de un entero combinación de las constantes especificadas a continuación que están definidas en el *ios_base.h*. Utilice el operador | (*or*) cuando necesite combinar dos o más constantes.

Los modos en los que se puede abrir el fichero son los mismos de la función **fopen** de C (véase el apéndice *La biblioteca estándar de C++*). En realidad, lo que hace C++ es utilizar la estructura **FILE** de C, evitándose así definir la semántica de **ios_base::(in, out, trunc, app, ate, binary)**. Para entender esto, lo mejor es ver el código fuente empleado para definir estos modos; este código es así:

```

switch (mode & (in|out|trunc|app|binary))
{
    // fichero de texto
    case ( out           ): return "w"; // escribir
    case ( out |app      ): return "a"; // añadir al final
    case ( out|trunc     ): return "w"; // escribir
    case ( in            ): return "r"; // leer
    case ( in|out        ): return "r+"; // leer y escribir
    case ( in|out|trunc  ): return "w+"; // escribir y leer

    // fichero binario
    case ( out |binary): return "wb"; // escribir
    case ( out |app|binary): return "ab"; // añadir al final
    case ( out|trunc |binary): return "wb"; // escribir
    case ( in |binary): return "rb"; // leer
    case ( in|out |binary): return "r+b"; // leer y escribir
    case ( in|out|trunc |binary): return "w+b"; // escribir y leer

    default: return 0; // modo inválido
}

```

■ Cuando estemos trabajando con un compilador C++ bajo el sistema operativo Windows hay que tener en cuenta las consideraciones descritas a continuación; por lo tanto, si es usuario de UNIX sáltese la letra pequeña. A diferencia de UNIX, en Windows un fichero puede ser abierto como fichero de *texto* o como fichero *binario*. La necesidad de dos formas diferentes es por las incompatibilidades existentes entre C y Windows ya que C fue diseñado originalmente para UNIX. Con dispositivos o ficheros de *texto*, el carácter ‘*n*’, utilizado en C++ para cambiar de línea, es traducido en dos caracteres (*CR+LF*) en una operación de salida y a la inversa, la combinación *CR+LF* es traducida en un único carácter ‘*n*’ (*LF*) cuando se trata de una entrada de datos. Esto significa que en Windows, cuando un programa C++ escribe en un fichero, traduce el carácter ‘*n*’ en los caracteres *CR+LF*; y cuando C++ lee desde un fichero y encuentra los caracteres *CR+LF*, los traduce a ‘*n*’; y cuando encuentra un *Ctrl+Z*, lo interpreta como un **eof** (carácter final de fichero). Esta traducción puede ocasionar problemas cuando nos desplazemos en el fichero un número de bytes determinado (método **seek?**). Para evitar este tipo de problemas utilice ficheros *binarios*, en los que las traducciones indicadas no tienen lugar.

■ En UNIX, la opción *binary* es ignorada aunque sintácticamente es aceptada. Esto permite la transportabilidad de un programa hecho en Windows a UNIX.

Cuando se especifica el modo *w* (texto o binario), si el fichero existe se destruye. Por ejemplo, el modo *w+* destruye el fichero si existe, en cambio *r+* no.

Si **open** se ejecuta satisfactoriamente y (*modo & ios_base::ate*) != 0, la posición de L/E se sitúa al final del fichero (igual que cuando se invoca a la función de C *std::fseek(file, 0, SEEK_END)*).

```
filebuf *close();
```

Este método vacía el *búfer* de salida, cierra el fichero y desconecta el fichero del objeto **filebuf** que recibe este mensaje. El método devuelve la dirección del objeto **filebuf** si la operación es satisfactoria o un valor **0** si ocurre un error.

```
bool is_open() const;
```

Este método devuelve un valor **true** si el objeto **filebuf** que recibe este mensaje está ligado a un fichero; esto es, verifica si el fichero está abierto. En otro caso devuelve un **0**.

El programa siguiente crea un búfer y lo vincula con un fichero abierto para escribir:

```
// filebuf.cpp
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
    char nombreFichero[30];
    filebuf buf; // declarar un búfer para un fichero

    cout << "Fichero: "; cin >> nombreFichero;
    // Abrir el fichero para escribir y asociarle el búfer
    if (buf.open(nombreFichero, ios::out) == 0)
    {
        cerr << "Error: no se puede abrir el fichero";
        return -1;
    }

    // ...
    cout << "El fichero " << nombreFichero;
    buf.is_open() ? cout << " está abierto\n"
                  : cout << " está cerrado\n";
}
```

Clase ostream

La clase **ostream** proporciona la funcionalidad necesaria para acceder secuencial o aleatoriamente a un fichero abierto para escribir. La mayor parte de los métodos de esta clase han sido heredados de la clase **ios**, de la que se deriva. Un objeto **ostream** tiene que ser conectado a otro de una clase derivada de **streambuf**, lo que supone construir un objeto **filebuf**. Quiere esto decir que las clases **ostream** y **streambuf** trabajan conjuntamente; la primera se encarga de dar formato a los da-

tos, y la segunda, de gestionar el búfer a través del cual se hace la transferencia de los datos. Para utilizar esta clase debe incluir en su código fuente la directriz:

```
#include <ostream>
```

La funcionalidad de esta clase está soportada por los siguientes métodos, entre otros (en la ayuda proporcionada en el CD puede ver todo acerca de ella):

```
ostream(streambuf *pbuf);
virtual ~ostream();
```

Estos dos métodos son el constructor y el destructor. El constructor crea un objeto **ostream** y lo conecta a un objeto de una clase derivada de **streambuf**, previamente construido, referenciado por *pbuf*.

```
ostream& put(char ch);
```

Este método inserta el carácter *ch* en el flujo que recibe el mensaje **put**. Si la operación falla, el método **setstate** activa el indicador **badbit**.

Recuerde que el estado de un flujo de E/S es modificado por el método **clear** siempre que una operación de entrada no tiene éxito. Por ejemplo, si una operación con **put** falla, **setstate** invoca a **clear** para que modifique el estado del flujo sobre el que se ejecuta **put**, poniendo la bandera de estado **badbit** a uno. Además, podríamos solicitar a **clear** que lanzara una excepción de la clase **failure** invocando al método **exceptions** de la clase **basic_ios**, así:

```
f.exceptions(ios::failbit | ios::badbit);
```

La sentencia anterior especifica que si sobre el flujo *f* ocurre un error que active **failbit** o **badbit**, se lanzará una excepción **ios_base::failure**.

```
ostream& write(const char *pch, int cont);
```

El método **write** inserta *cont* caracteres de la matriz *pch* en el flujo que recibe el mensaje **write**. Si la operación falla, se activa el indicador **badbit**. A diferencia del operador de inserción (<<), este método produce una salida en binario, no en ASCII.

```
ostream& flush();
```

Este método envía el contenido del búfer asociado con el flujo al fichero que está vinculado con el mismo. Retorna ***this**. Si *rdbuf()* devuelve un puntero nulo, no modifica el estado del flujo; en otro caso, llama a *rdbuf()->pubsync()*, y si éste retorna -1, activa **badbit**.

```
pos_type tellp()
```

Obtiene la posición actual de escritura.

```
ostream& seekp(pos_type pos)
ostream& seekp(off_type des, ios_base::seekdir pos)
```

La primera versión de este método cambia la posición actual de escritura a la posición *pos* de carácter del fichero, y la segunda desplaza la posición actual de escritura *des* caracteres respecto de la posición *pos* que puede ser: **beg** (principio del fichero), **cur** (posición actual) o **end** (final del fichero). Si la operación falla, se activa el indicador **failbit**.

```
ostream& operator<<(...)
```

Se trata del operador de inserción. Este método produce una salida en ASCII.

Según lo expuesto anteriormente, cuando utilizamos un objeto **ostream** para escribir en un fichero en el disco, primero tenemos que construir un objeto **filebuf** con el fin de suministrar un búfer para el fichero. Por ejemplo, el siguiente fragmento de código lee una cadena de caracteres de la entrada estándar y la escribe en un fichero en disco:

```
string nombreFichero;
filebuf buf; // declarar un búfer para un fichero
cout << "Fichero: "; getline(cin, nombreFichero);
// Abrir el fichero para escribir y asociarle búfer
buf.open(nombreFichero.c_str(), ios::out);
ostream os(&buf); // vincular el búfer con el stream os
// Escribir datos en el fichero
string linea;
getline(cin, linea); // leer una línea de la entrada estándar
os << linea << endl; // escribir la línea finalizada con '\n'
```

No obstante, la forma más normal de proceder no es ésta, sino utilizar objetos de las clases **ofstream** para ficheros en el disco y de **ostreamstream** para matrices de caracteres.

Clase istream

La clase **istream** proporciona la funcionalidad necesaria para acceder secuencial o aleatoriamente a un fichero abierto para leer. La mayor parte de los métodos de esta clase han sido heredados de la clase **ios**, de la que se deriva. Un objeto **istream** tiene que ser conectado a otro de una clase derivada de **streambuf**, lo que supone construir un objeto **filebuf**. Quiere esto decir que las clases **istream** y **streambuf**

trabajan conjuntamente; la primera se encarga de dar formato a los datos y la segunda, de gestionar el búfer a través del cual se hace la transferencia de los datos. Para utilizar esta clase, debe incluir en su código fuente la directriz:

```
#include <istream>
```

La funcionalidad de esta clase está soportada por los siguientes métodos, entre otros (en la ayuda proporcionada en el CD puede ver todo acerca de ella):

```
istream(streambuf *pbuf);
virtual ~istream();
```

Estos dos métodos son el constructor y el destructor. El constructor crea un objeto **istream** y lo conecta a un objeto de una clase derivada de **streambuf**, previamente construido, referenciado por *pbuf*.

```
istream& get(char& ch);
```

El método **get** extrae un carácter del flujo que recibe el mensaje **get** y lo almacena en *ch*. Observe que el método devuelve una referencia al flujo; si la operación falla, se activa el indicador **failbit**, y si además se alcanza el final del fichero, se activa el indicador **eofbit**.

```
istream& getline(char *ch, int cont, char delim = '\n');
```

El método **getline** extrae una cadena de caracteres del flujo que recibe este mensaje. La terminación '\0' es añadida automáticamente a la cadena leída. Se entiende por cadena desde la posición actual en el fichero hasta el delimitador *delim* (el delimitador se extrae, pero no se almacena) hasta el final del fichero, o hasta que el número de caracteres extraídos sea igual a *cont-1*. Si la operación falla, se activa el indicador **failbit**.

```
istream& read(char *ch, int cont);
```

Este método extrae una cadena de caracteres del flujo que recibe este mensaje. A la cadena no se le añade el carácter de terminación nulo. Se entiende por cadena desde la posición actual en el fichero hasta el final del fichero o hasta que el número de caracteres extraídos sea igual a *cont*. En contraposición al operador de extracción (>>) que trabaja con ficheros de texto, este método es útil cuando se trabaja con ficheros binarios (fueron escritos con **write**). Si en una operación de lectura se alcanza el fin de fichero, se activan los indicadores **failbit** y **eofbit**.

```
istream& ignore(int cont = 1, int delim = char_traits::eof());
```

El método **ignore** extrae y descarta una cadena de hasta *cont* caracteres del flujo que recibe este mensaje. La extracción finalizará cuando se hayan extraído *cont* caracteres, cuando se alcance el final del fichero o cuando se extraiga el carácter delimitador, *delim* (**char_traits::eof**()) devuelve **EOF**.

```
int peek() const;
```

Este otro método devuelve el siguiente carácter del flujo que recibe el mensaje **peek**, sin extraerlo. Si el carácter se corresponde con el final del fichero, entonces el método devuelve **EOF**.

```
int gcount() const;
```

Este método devuelve el número de caracteres que fueron extraídos en la última operación de lectura sin formato (las lecturas con el operador de extracción son con formato) realizada sobre el flujo que recibe este mensaje.

```
istream& unget();
```

Devuelve al flujo el último carácter extraído. Si la operación falla, se activa el indicador **badbit**.

```
istream& putback(char c);
```

Devuelve al flujo el carácter *c*. Si la operación falla, se activa el indicador **badbit**.

```
pos_type tellg();
```

Obtiene la posición actual de lectura.

```
istream& seekg(pos_type pos)
istream& seekg(off_type des, ios_base::seekdir pos)
```

Cambia la posición actual de lectura (la explicación de los parámetros fue expuesta anteriormente en la clase **ostream**). Si la operación falla, se activa el indicador **failbit**.

```
istream& operator>>(...)
```

Se trata del operador de extracción. Este método es el adecuado para leer datos escritos con el operador de inserción expuesto anteriormente. Hay que tener presente que este operador interpreta el espacio en blanco como un separador. Si necesita leer cualquier carácter, entonces, utilice **getline**.

Según lo expuesto, cuando utilizamos un objeto **istream** para leer de un fichero en el disco, primero tenemos que construir un objeto **filebuf** con el fin de suministrar un búfer para el fichero. Por ejemplo, el siguiente programa lee cadenas de caracteres de un fichero en disco y las escribe en la salida estándar.

```
string nombreFichero;
filebuf buf; // declarar un búfer para un fichero
cout << "Fichero: "; getline(cin, nombreFichero);
// Abrir el fichero para leer y asociarle un búfer
buf.open(nombreFichero.c_str(), ios::in);
istream is(&buf); // vincular el búfer con el stream is
// Leer datos del fichero
string linea;
getline(is, linea); // leer una línea del fichero
cout << linea << endl; // mostrar la línea finalizada con '\n'
```

No obstante, la forma más normal de proceder no es ésta, sino utilizar objetos de las clases **ifstream** para ficheros en el disco y de **istringstream** para matrices de caracteres.

Clase **istream**

La clase **istream** proporciona la funcionalidad necesaria para acceder secuencial o aleatoriamente a un fichero abierto para leer o escribir. Está derivada de las clases **ostream** e **istream**, razón por la cual hereda toda la funcionalidad de éstas.

La clase **istream** trabaja conjuntamente con las clases derivadas de **streambuf**; por ejemplo, con **filebuf**. Esto quiere decir que cuando utilizamos un objeto **istream** para leer o escribir en un fichero en el disco, primero tenemos que construir un objeto **filebuf** con el fin de suministrar un búfer para el fichero. No obstante, la forma más normal de proceder no es ésta, sino utilizar objetos de las clases **fstream** para ficheros en el disco y de **stringstream** para matrices de caracteres.

Para utilizar esta clase debe incluir en su código fuente la directriz:

```
#include <istream>
```

Esta clase, además de la funcionalidad heredada, tiene un constructor:

```
istream(streambuf* pbuf);
```

el cual construye un objeto **istream** y lo conecta a un objeto de una clase derivada de **streambuf**, previamente construido, referenciado por *pbuf*.

Clase ofstream

La clase **ofstream** es una clase derivada de **ostream** especializada en manipular ficheros en el disco abiertos para escribir. A diferencia de lo que sucedía con **ostream**, cuando se construye un objeto de esta clase, el constructor lo conecta automáticamente con un objeto **filebuf** (un búfer). Para utilizar esta clase debe incluir en su código fuente la directriz:

```
#include <fstream>
```

La funcionalidad de esta clase está soportada por los siguientes métodos, entre otros (en la ayuda proporcionada en el CD puede ver todo acerca de ella):

```
ofstream(const char *nombre_fichero,
         ios_base::openmode modo = ios::out | ios::trunc);
```

Este método es el constructor de la clase; también existen un constructor sin parámetros y un destructor. En la clase **filebuf** expuesta anteriormente en este mismo capítulo, puede ver una descripción de cada uno de los parámetros. Un ejemplo de cómo se utiliza este método es el siguiente:

```
ofstream os("texto01.txt");
if (!os)
    throw "No se puede abrir el fichero";
```

Observe, en el constructor, que un **ofstream** se abre por omisión para escribir.

```
void open(const char *nombre_fichero,
         ios_base::openmode modo = ios::out | ios::trunc);
```

Este método abre el fichero especificado por *nombre_fichero* y lo conecta con el objeto **filebuf** del **ofstream** que recibe el mensaje **open**. Para comprobar si ha ocurrido un error, verifique el estado del flujo. La descripción de los parámetros es la misma que para el constructor. A continuación se muestra un ejemplo:

```
ofstream os;
os.open("texto01.txt");
if (!os)
    throw "No se puede abrir el fichero";
```

```
void close();
```

Este método llama al método **filebuf::close**.

```
int is_open() const;
```

Este método devuelve un valor distinto de 0 si el objeto **ofstream** que recibe este mensaje está ligado a un fichero; esto es, verifica si el fichero está abierto. En otro caso devuelve un 0.

```
filebuf *rdbuf() const;
```

Este método devuelve un puntero al búfer vinculado con el flujo.

El siguiente programa define varias variables de diferentes tipos, las inicia con unos valores determinados (podríamos haberlas leído desde la entrada estándar) y las escribe en el fichero denominado *datos.dat*:

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
    char cDato = 'A';
    int iDato = 1234;
    double dDato = 3.141592;
    string s1Dato = "una cadena";
    string s2Dato = "otra cadena";

    // Abrir un fichero para escribir
    string nombreFichero = "datos.dat";
    ofstream ofs(nombreFichero.c_str());

    // Verificar si ocurrió un error
    if (!ofs.good())
    {
        cout << "Error: no se puede abrir el fichero "
              << nombreFichero << " para escribir.\n";
        return 0;
    }

    // Escribir los datos en el fichero (dos veces)
    for (int i = 0; i < 2; i++)
    {
        // Escribir un registro
        ofs << s1Dato << endl;
        ofs << cDato << endl;
        ofs << iDato << endl;
        ofs << dDato << endl;
        ofs << s2Dato << endl;
    }
}
```

```

// Cerrar el fichero
ofs.close();
return 0;
}

```

Clase ifstream

La clase **ifstream** es una clase derivada de **istream** especializada en manipular ficheros en el disco abiertos para leer. A diferencia de lo que sucedía con **istream**, cuando se construye un objeto de esta clase, el constructor lo conecta automáticamente con un objeto **filebuf** (un búfer). Para utilizar esta clase debe incluir en su código fuente la directriz:

```
#include <fstream>
```

La funcionalidad de esta clase está soportada por los siguientes métodos, entre otros (en la ayuda proporcionada en el CD puede ver todo acerca de ella):

```
ifstream(const char *nombre_fichero,
         ios_base::openmode modo = ios::in);
```

Este método es el constructor de la clase; también existen un constructor sin parámetros y un destructor. En la clase **filebuf** expuesta anteriormente en este mismo capítulo, puede ver una descripción de los parámetros. Un ejemplo de cómo se utiliza este método es el siguiente:

```
ifstream is("texto01.txt");
if (!is)
    throw "No se puede abrir el fichero";
```

Observe, en el constructor, que un **ifstream** se abre por defecto para leer.

```
void open(const char *nombre_fichero,
         ios_base::openmode modo = ios::in);
```

El método **open** abre el fichero especificado por *nombre_fichero* y lo conecta con el objeto **filebuf** del **ifstream** que recibe el mensaje **open**. Para comprobar si ha ocurrido un error, verifique el estado del flujo. La descripción de los parámetros es la misma que para el constructor. Por ejemplo:

```
ifstream is;
is.open("texto01.txt")
if (!is)
    throw "No se puede abrir el fichero";
```

Este ejemplo abre el fichero *texto01.txt* para leer. Si el fichero no existe, se produce un error.

```
void close();
```

Este método llama al método **filebuf::close**.

```
int is_open() const;
```

El método **is_open** devuelve un valor distinto de 0 si el objeto **ifstream** que recibe este mensaje está ligado a un fichero; esto es, verifica si el fichero está abierto. En otro caso devuelve un 0.

```
filebuf *rdbuf() const;
```

Este método devuelve un puntero al búfer vinculado con el flujo.

El siguiente programa define las variables de los tipos correspondientes a los datos almacenados por cada registro que deseamos leer del fichero, lee, utilizando el operador >>, cada uno de los registros almacenados en el fichero y muestra los datos leídos en la salida estándar (véase el ejemplo anterior, donde se construyó el fichero *datos.dat*):

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

using namespace std;

int main()
{
    char cDato;
    int iDato;
    double dDato;
    string s1Dato;
    string s2Dato;

    // Abrir un fichero para leer
    string nombreFichero = "datos.dat";
    ifstream ifs(nombreFichero.c_str());

    // Verificar si ocurrió un error
    if (!ifs.good())
    {
        cout << "Error: no se puede abrir el fichero "
              << nombreFichero << " para leer.\n";
        return 0;
    }
}
```

```

// Leer los datos del fichero
do
{
    // Leer un registro
    getline(ifs, s1Dato);
    if (ifs.eof()) break; // no hay más datos
    ifs >> cDato;
    ifs >> iDato;
    ifs >> dDato;
    ifs.ignore(); // eliminar el endl
    getline(ifs, s2Dato);
    // Mostrar los datos leídos
    cout << s1Dato << endl
         << cDato << endl
         << iDato << endl
         << dDato << endl
         << s2Dato << endl << endl;
}
while (true);

// Cerrar el fichero
ifs.close();

return 0;
}

```

Clase `fstream`

La clase **`fstream`** es una clase derivada de **`iostream`** especializada en manipular ficheros en el disco abiertos para leer y/o escribir. A diferencia de lo que sucedía con **`iostream`**, cuando se construye un objeto de esta clase, el constructor lo conecta automáticamente con un objeto **`filebuf`** (un búfer). Para utilizar esta clase debe incluir en su código fuente la directriz:

```
#include <fstream>
```

La funcionalidad de esta clase está soportada por los siguientes métodos, entre otros (en la ayuda proporcionada en el CD puede ver todo acerca de ella):

```
fstream(const char *nombre_fichero,
        ios_base::openmode modo = ios::in | ios::out);
```

Este método es el constructor de la clase; su segundo parámetro indica que, por omisión, el fichero será abierto para leer y escribir, esto es, si el fichero existe no se destruye. También existe un constructor sin parámetros y un destructor. En la clase **`filebuf`** expuesta anteriormente en este mismo capítulo, puede ver una

descripción de los parámetros. Un ejemplo de cómo se utiliza este método es el siguiente:

```
fstream fs("texto.txt");
if (!fs)
    throw "No se puede abrir el fichero";
```

Observe que el objeto *fs* vinculado al fichero *texto.txt* permite leer y escribir sobre dicho fichero. Si el fichero *texto.txt* no existe, se produce un error, ya que se ha especificado el modo **in|out** (los modos fueron expuestos al hablar de **filebuf**).

```
void open(const char *nombre_fichero,
          ios_base::openmode modo = ios::in | ios::out);
```

Este otro método abre el fichero especificado por *nombre_fichero* y lo conecta con el objeto **filebuf** del **fstream** que recibe el mensaje. Por ejemplo, las siguientes líneas de código realizan la misma función que el ejemplo anterior:

```
fstream fs; // construye el flujo fs sin abrir el fichero
fs.open("texto.txt");
if (!fs)
    throw "No se puede abrir el fichero";
```

Para comprobar si ha ocurrido un error, verifique el estado del flujo. En la clase **filebuf** puede ver una descripción de los parámetros.

```
void close();
```

Este método llama al método **filebuf::close**.

```
int is_open() const;
```

Este método devuelve un valor distinto de 0 si el objeto **fstream** que recibe este mensaje está ligado a un fichero; esto es, verifica si el fichero está abierto. En otro caso devuelve un 0.

```
filebuf *rdbuf() const;
```

Este método devuelve un puntero al búfer vinculado con el flujo.

El siguiente ejemplo abre un fichero de texto para escribir y leer, escribe en él nombres de provincias almacenados en una matriz, para, finalmente, leer esta información del fichero y mostrarla por pantalla. Leer la información exigirá situar la posición de L/E al comienzo del fichero. También, a la hora de leer los nombres, dese cuenta que los hay que incluyen espacios en blanco. Si el fichero no puede abrirse, se lanzará una excepción de tipo **const char *** indicándolo.

```

#include <fstream>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    try
    {
        char *provincia[] = {"Madrid", "Santander", "Sevilla",
                             "A Coruña", "Valencia"};

        // Abrir el fichero para escribir y leer (w+)
        fstream fs;
        string nombreFichero = "datos";
        fs.open(nombreFichero.c_str(), ios::out|ios::trunc|ios::in);

        if (!fs) throw "No se puede abrir el fichero";

        // Escribir en el fichero
        for (int i = 0; i < sizeof(provincia)/sizeof(char *); ++i)
            fs << provincia[i] << "\n";

        // Leer del fichero
        string str;
        fs.seekg(ios::beg);
        for (int i = 0; i < sizeof(provincia)/sizeof(char *); ++i)
        {
            getline(fs, str);
            // fs >> str; // interpretaría el espacio como separador
            cout << str << endl;
        }
    }
    catch (const char * str)
    {
        cout << str << endl;
    }
}

```

E/S UTILIZANDO REGISTROS

Los datos pueden ser escritos y leídos en bloques denominados registros con los métodos **write** y **read** heredados por **fstream** indirectamente de las clases **ostream** e **istream**. Entendemos por registro un conjunto de datos de longitud fija, tales como estructuras o elementos de una matriz; en general, objetos. No obstante, aunque lo más habitual en C++ sea que un registro se corresponda con una estructura de datos, es factible también que un registro se corresponda con una variable

de tipo **char**, **int**, **float**, con una cadena de caracteres, con un grupo de ellas o con un objeto de una clase, entre otros.

Según lo estudiado, el método **write** permite escribir n bytes almacenados en un búfer referenciado por un puntero a **char**, en el fichero vinculado con el flujo que recibe ese mensaje. Veamos un ejemplo:

```
struct t_tfno
{
    char nombre[30];
    char direccion[40];
    long telefono;
};

t_tfno persona; // estructura de tipo t_tfno
int tam = sizeof(t_tfno);
// Abrir el fichero para escribir
ofstream ofs("lista.tfno");
// Leer los datos
cout << "Nombre:    "; cin.getline(persona.nombre, 30);
cout << "Dirección: "; cin.getline(persona.direccion, 40);
cout << "Teléfono:  "; cin >> persona.telefono;
// Escribir el tamaño del registro y su contenido
ofs.write(reinterpret_cast<char *>(&tam), sizeof(int));
ofs.write(reinterpret_cast<char *>(&persona), sizeof(persona));
// Cerrar el fichero
ofs.close();
```

El método **write** almacena los datos numéricos en formato binario. Esto quiere decir que, en un sistema de 32 bits, un **int** ocupa cuatro bytes, un **float** ocupa cuatro bytes, un **double** ocupa ocho bytes, etc.

- En Windows, no hay que confundir el formato binario empleado para almacenar un dato numérico con la forma binario (*binary*) en la que se puede abrir un fichero para evitar que ocurra la traducción entre los caracteres ‘\n’ y CR+LF.

También hemos estudiado que el método **read** permite leer n bytes (almacenados por **write**) del fichero vinculado con el flujo que recibe ese mensaje y almacenarlos en un búfer referenciado por un puntero a **char**. Veamos un ejemplo:

```
t_tfno persona; // estructura de tipo t_tfno
int tam = sizeof(t_tfno);
// Abrir el fichero para leer
ifstream ifs("lista.tfno");
// Leer el tamaño del registro y su contenido
ifs.read(reinterpret_cast<char *>(&tam), sizeof(int));
ifs.read(reinterpret_cast<char *>(&persona), sizeof(persona));
// Mostrar los datos leídos
```



```

cout << "Tamaño: " << tam << endl;
cout << "Nombre: " << persona.nombre << endl;
cout << "Dirección: " << persona.direccion << endl;
cout << "Teléfono: " << persona.telefono << endl;
// Cerrar el fichero
ifs.close();

```

Los ejemplos expuestos demuestran que los métodos **write** y **read** permiten escribir y leer, respectivamente, variables de tipo **char**, **int**, **float**, matrices, estructuras, etc. Esto es, pueden reemplazar perfectamente a otros métodos de E/S.

ESCRIBIR DATOS EN LA IMPRESORA

La salida de un programa puede también ser enviada a un dispositivo de salida que no sea el disco o la pantalla; por ejemplo, a una impresora conectada al puerto paralelo. Si su sistema no tiene definido un flujo estándar para el puerto paralelo, la solución es definir uno y vincularlo a dicho dispositivo.

Una forma de realizar lo expuesto es crear un flujo hacia el dispositivo *lpt1*, *lpt2* o *prn* y escribir en ese flujo (los nombres indicados son los establecidos por Windows para nombrar a la impresora; en UNIX la primera impresora tiene asociado el nombre */dev/lp0*, la segunda */dev/lp1*, etc.). Las siguientes líneas de código muestran cómo crear un flujo hacia una impresora:

```

fstream impre("lpt1", ios::out);
// en UNIX/LINUX utilizar "/dev/lp0"
if (!impre)
    cerr << "La impresora no está lista\n";
impre << "Esta cadena se escribe en la impresora\n";
impre << "\f"; // avanzar una página

```

ABRIENDO FICHEROS PARA ACCESO SECUENCIAL

El tipo de acceso más simple a un fichero de datos es el secuencial: los registros que se escriben en el fichero son colocados automáticamente uno a continuación de otro y, cuando se leen, se empieza por el primero, se continúa con el siguiente, y así sucesivamente hasta alcanzar el final. Esta forma de proceder posibilita que los registros puedan ser de cualquier longitud, incluso de un solo byte.

Este tipo de acceso generalmente se utiliza con ficheros de texto en los que se escribe toda la información desde el principio hasta el final y se lee de la misma forma. En cambio, los ficheros de texto no son los más apropiados para almacenar grandes series de números, porque cada número es almacenado como una secuencia de bytes; esto significa que un número entero de nueve dígitos ocupa nueve

bytes en lugar de los cuatro requeridos para un entero. De ahí que para el tratamiento de información numérica se sugiera utilizar **write** y **read**.

Un ejemplo de acceso secuencial

Después de la teoría expuesta hasta ahora acerca del trabajo con ficheros, habrá observado que la metodología de trabajo se repite. Es decir, para escribir datos en un fichero:

- Definimos un flujo hacia el fichero en el que deseamos escribir datos.
- Leemos los datos del dispositivo de entrada o de otro fichero y los escribimos en nuestro fichero. Este proceso se hace normalmente registro a registro. Para ello, utilizaremos los métodos proporcionados por la interfaz del flujo.
- Cerramos el flujo.

Para leer datos de un fichero existente:

- Abrimos un flujo desde el fichero del cual queremos leer los datos.
- Leemos los datos del fichero y los almacenamos en variables de nuestro programa con el fin de trabajar con ellos. Este proceso se hace normalmente registro a registro. Para ello, utilizaremos los métodos proporcionados por la interfaz del flujo.
- Cerramos el flujo.

Esto pone de manifiesto que un fichero no es más que un medio permanente de almacenamiento de datos, dejando esos datos disponibles para cualquier programa que necesite manipularlos. Lógicamente, los datos serán recuperados del fichero con el mismo formato con el que fueron escritos, de lo contrario los resultados serán inesperados. Es decir, si los datos fueron guardados en el orden: un **int** y una estructura *t_tfno*, tendrán que ser recuperados en este orden y con este mismo formato. Sería un error recuperar primero una estructura *t_tfno* y después un **int**, los resultados serían inesperados.

Como ejemplo de lo que acabamos de exponer, vamos a realizar un programa que permita crear un fichero nuevo, añadir información a uno existente o buscar el contenido de un registro. El nombre del fichero será introducido a través del teclado. Cada registro del fichero estará formado por los datos *referencia* y *precio*. Así mismo, para que el usuario pueda elegir cualquiera de las opciones enunciadas, el programa visualizará en pantalla un menú similar al siguiente:

Fichero actual: ninguno

1. Nuevo fichero
 2. Abrir fichero
 3. Añadir registros
 4. Buscar un registro
 5. Salir
-

Opción (1 - 5): 1

Nombre del fichero: articulos.dat

El fichero articulos.dat existe. ¿Desea sobrescribirlo? (s/n): s

Fichero actual: articulos.dat abierto para añadir

1. Nuevo fichero
 2. Abrir fichero
 3. Añadir registros
 4. Buscar un registro
 5. Salir
-

Opción (1 - 5): 3

Introducir datos. Para finalizar, responder "fin" a Referencia:.

Referencia: 123a7x

Precio: 150.25

...

Referencia: fin

Fichero actual: articulos.dat abierto para añadir

1. Nuevo fichero
 2. Abrir fichero
 3. Añadir registros
 4. Buscar un registro
 5. Salir
-

Opción (1 - 5): 2

Nombre del fichero: articulos.dat

1. leer
 2. añadir
-

Opción (1 - 2): 1

Fichero actual: articulos.dat abierto para leer

1. Nuevo fichero
 2. Abrir fichero
 3. Añadir registros
 4. Buscar un registro
 5. Salir
-

Opción (1 - 5): 4

Referencia: 123a7x

precio: 150.25

Fichero actual: articulos.dat abierto para leer

1. Nuevo fichero
 2. Abrir fichero
 3. Añadir registros
 4. Buscar un registro
 5. Salir
-

Opción (1 - 5):

Para que el problema planteado sea fácil de resolver, vamos a crear una interfaz que responda a las opciones presentadas por el menú. Para ello, escribiremos una clase *CRegistro* que permita manipular cada registro individual del fichero y una clase *CArticulos* derivada de la clase **fstream**, con una interfaz pública que permita realizar las operaciones indicadas en el menú. Esto es, intentemos aprovechar las ventajas que ofrece la programación orientada a objetos para, utilizando la interfaz proporcionada por **fstream**, crear otra interfaz más sencilla que le permita al usuario escribir un programa como el siguiente:

```
// test.cpp - Acceso secuencial
#include <iostream>
#include <fstream>
#include <string>
#include "registro.h"
```

```
#include "articulos.h"
#include "leerdatos.h"
using namespace std;

int main()
{
    // Opciones del menú
    static char *opciones[] =
    {
        "Nuevo fichero",
        "Abrir fichero",
        "Añadir registros",
        "Buscar un registro",
        "Salir"
    };
    int nOpciones = sizeof(opciones)/sizeof(char*);
    string cadenabuscar, nombreFichero, str;

    CArticulos fs; // flujo de E/S
    float precio = 0;
    bool salir = false;

    while (true)
    {
        str = "ninguno";
        if (fs.Modos() == 1)
            str = nombreFichero + " abierto para leer";
        else if (fs.Modos() == 2)
            str = nombreFichero + " abierto para añadir";
        cout << "\nFichero actual: " << str;
        try
        {
            switch (menu(opciones, nOpciones))
            {
                case 1: // crear un fichero nuevo
                    cout << "\nNombre del fichero: ";
                    leerDato(nombreFichero);
                    fs.Nuevo(nombreFichero);
                    break;

                case 2: // abrir un fichero existente
                    cout << "\nNombre del fichero: ";
                    leerDato(nombreFichero);
                    fs.Abrir(nombreFichero);
                    break;
                case 3: // añadir un registro al fichero actual
                    fs.Agregar();
                    break;

                case 4: // buscar un registro en el fichero actual
                    cout << "Referencia: ";
```

```

        leerDato(cadenabuscar);
        precio = fs.Buscar(cadenabuscar);
        if (precio == -1)
            cout << "búsqueda fallida\n";
        else
            cout << "precio: " << precio << endl;
        break;

    case 5: // salir
        salir = true;
    }
}
catch(ios_base::failure& e)
{
    cout << e.what() << endl;
}
if (salir) break;
}
}

```

Del estudio del programa anterior se deduce que éste utiliza:

- La interfaz proporcionada por la plantilla y las funciones almacenadas en los ficheros *leerdatos.h* y *leerdatos.cpp* que escribimos en el capítulo anterior, y que permitían crear un menú y leer de forma segura cualquier tipo de datos que pueda ser leído por el método **operator>>** de **cin**, además de datos de tipo **string**.
- La función *menú* para presentar un menú con las opciones disponibles.
- Y la interfaz de una clase *CARTICULO* que da respuesta a las opciones presentadas por el menú. Un objeto de esta clase representará un fichero almacenado en el disco y sus registros podrán ser manipulados a través de la interfaz de otra clase *CRegistro*.

Según lo expuesto, la funcionalidad de la clase *CRegistro* estará soportada por los atributos *referencia* y *precio* y por los siguientes métodos:

- Un constructor que inicie por omisión los atributos a cero.
- El método *AsignarRegistro* para almacenar datos en un registro.
- Y los métodos *ObtenerReferencia* y *ObtenerPrecio*, que permitan obtener los datos de un registro.

La declaración de esta clase puede ser de la forma siguiente:

```

// registro.h - Declaración de la clase CRegistro
#ifdef !defined(_REGISTRO_H_)

```


Si siguiendo con el ejemplo, la funcionalidad de la clase *CArticulos* estará soportada por el atributo *modo*, con el fin de almacenar el modo 1 (leer) o 2 (añadir) en el que se abra el fichero con el que se desee trabajar, y por los siguientes métodos:

- Un constructor sin argumentos que permita iniciar el atributo *modo* a cero, lo que indicará que no hay ningún fichero abierto.
- Un destructor que permita cerrar el fichero cuando el flujo vinculado con el mismo deje de existir.
- Los métodos *Nuevo* para crear un nuevo fichero abierto para añadir, *Abrir* para abrir un fichero para leer o añadir, *Agregar* para abrir un fichero existente para añadir nuevos registros, *Buscar* para presentar en pantalla el precio de un determinado artículo, *Existe* para comprobar si un fichero existe y *Modo* para obtener el modo en el que se abrió el fichero.

```
// articulos.h - Declaración de la clase CArticulos
#if !defined( _ARTICULOS_H_ )
#define _ARTICULOS_H_

#include <fstream>

////////////////////////////////////
// clase base de datos
class CArticulos : private std::fstream
{
private:
    int modo; // 1 = leer, 2 = añadir
public:
    CArticulos();
    ~CArticulos();
    void Nuevo(std::string&); // crear un fichero
    void Abrir(std::string&); // abrir un fichero
    void Agregar(); // añadir registros al fichero
    float Buscar(std::string&); // buscar un registro
    bool Existe(std::string&); // true si el fichero existe
    int Modo() { return modo; }
};

#endif // _ARTICULOS_H_
////////////////////////////////////
```

Los métodos *Nuevo*, *Abrir* y *Agregar* lanzarán una excepción del tipo **failure** siempre que alguna acción sobre el flujo vinculado con el fichero abierto dé lugar a que se active alguno de los indicadores **failbit** o **badbit**. El método *Buscar* no lanza ninguna excepción, simplemente devuelve el *precio* del artículo buscado o -1

si la búsqueda falla. El método *Existe* devuelve **true** si un determinado fichero existe y **false** en caso contrario. Y el método *Modo* devuelve el valor del atributo *modo*: 1 si el fichero fue abierto para leer o bien 2 si el fichero fue abierto para añadir registros.

El método *Nuevo* comprobará si el fichero que se trata de crear existe; si existe, preguntará al usuario si desea sobrescribirlo, y en caso afirmativo eliminará todos sus registros. En cualquier caso, finalmente el fichero será abierto para añadir. Esta opción podrá ejecutarse tantas veces como el usuario estime oportuno; por lo tanto, cada vez que el método se ejecute comprobará si ya hay un fichero abierto, en cuyo caso lo cerrará.

```
void CArticulos::Nuevo(string& nomf)
{
    // Habilitar las excepciones del tipo failure
    clear(); // desactivar los indicadores que haya activos
    exceptions(ios::failbit | ios::badbit);

    // Crear un nuevo fichero
    if (nomf.empty())
        throw failure(string("Especificar un nombre para el fichero"));

    if (is_open()) close(); // cerrar el fichero si está abierto
    modo = 0;
    // Comprobar si el fichero existe
    if (Existe(nomf))
    {
        cout << "El fichero " << nomf << " existe. "
              << "¿Desea sobrescribirlo? (s/n): ";
        string resp("n");
        leerDato(resp);
        if (resp == "s")
        {
            // Eliminar todos sus registros
            open(nomf.c_str(), ios::out | ios::trunc);
            close();
        }
    }
    // Abrir el fichero para añadir registros
    open(nomf.c_str(), ios::out | ios::app | ios::binary);
    modo = 2; // añadir
}
```

El método *Abrir* presentará un menú que permitirá al usuario decidir si quiere abrir el fichero para leer o para añadir. Un mismo fichero podrá ser abierto tantas veces como sea requerido; por lo tanto, cada vez que se solicite esta acción, el método comprobará si ya hay un fichero abierto, en cuyo caso lo cerrará.

```
void CArticulos::Abrir(string& nomf)
{
    // Habilitar las excepciones del tipo failure
    clear(); // desactivar los indicadores que haya activos
    exceptions(ios::failbit | ios::badbit);

    // Abrir un fichero
    if (nomf.empty())
        throw failure(string("Especificar un nombre para el fichero"));

    if (is_open()) close(); // cerrar el fichero si está abierto
    modo = 0;
    // Opciones del menú
    static char *opciones[] =
    {
        "leer",
        "añadir",
    };
    int op, nOpciones = sizeof(opciones)/sizeof(char*);

    if ((op = menu(opciones, nOpciones)) == 1) // leer
        open(nomf.c_str(), ios::in | ios::binary);
    else // añadir
        open(nomf.c_str(), ios::out | ios::app | ios::binary);
    modo = op;
}
```

El método *Agregar* sólo se podrá ejecutar si el fichero fue abierto para añadir. Permite añadir uno o más registros al final del fichero. Los datos *referencia* y *precio* para cada registro los solicitará del usuario, el cual indicará que no quiere añadir más registros introduciendo la *referencia* "fin".

```
void CArticulos::Agregar()
{
    // Habilitar las excepciones del tipo failure
    clear(); // desactivar los indicadores que haya activos
    exceptions(ios::failbit | ios::badbit);

    // Escribir registros en un fichero
    if (modo != 2)
        throw failure(string("Abrir el fichero para añadir"));

    CRegistro reg;
    int tm = sizeof(reg);
    char ref[30];
    float pre;
    cout << "\nIntroducir datos. Para finalizar, "
        << "responder \"fin\" a Referencia:.\n\n";
    while(true)
    {
```

```

    cout << "Referencia: "; leerDato(ref);
    if (!strcmp(ref, "fin")) break;
    cout << "Precio:      "; leerDato(pre);

    reg.AsignarRegistro(ref, pre);
    write(reinterpret_cast<char *>(&reg), tm);
}
}

```

El método *Buscar* recibirá como parámetro la referencia que se desea buscar y devolverá el *precio* del artículo correspondiente, o bien -1 si la referencia no existe. Esta forma de proceder hace que haya que deshabilitar las excepciones de tipo **failure**, porque si se alcanza el final del fichero, se activará tanto el indicador **eofbit** como **failbit**, lo que daría lugar a que se lanzara una excepción que impediría ejecutar la sentencia **return**.

```

float CArticulos::Buscar(string& str)
{
    // Deshabilitar las excepciones del tipo failure
    exceptions(ios::goodbit);

    // Buscar un registro en el fichero
    if (modo != 1)
    {
        cout << "Abrir el fichero para leer\n";
        return -1;
    }
    CRegistro reg;
    seekg(0, ios::beg); // situarse al principio
    int tm = sizeof(reg);
    char ref[30];
    // Buscar un artículo y devolver el precio
    string refe;
    if (str.empty()) return false;
    while (read(reinterpret_cast<char *>(&reg), tm))
    {
        // Buscar por la referencia
        refe = string(reg.ObtenerReferencia(ref));
        if (refe.empty()) continue;
        // ¿str es la referencia?
        if (str == refe)
            return reg.ObtenerPrecio();
    }
    clear(); // desactivar eofbit y failbit
    return -1;
}

```

El método *Existe* recibirá como parámetro el nombre del fichero del que queremos saber si existe. Un intento de abrirlo para leer responderá a tal pregunta, ya que si el fichero no existe se activará el indicador **failbit**.

```
bool CArticulos::Existe(string& nombf)
{
    // Deshabilitar las excepciones del tipo failure
    exceptions(ios::goodbit);

    // Intentar abrir el fichero para leer
    open(nombf.c_str(), ios::in);
    if (fail())
    {
        clear();
        return false;
    }
    close();
    return true;
}
```

El método *Modo* fue implementado en la declaración de la clase y simplemente retorna el valor del atributo *modo*.

Observe también cómo los métodos de la clase **fstream**, por ejemplo **open**, **write**, **read**, etc. son invocados directamente por los métodos de la clase *CArticulos*, puesto que los ha heredado. Cuando uno de estos métodos se ejecuta, ¿sobre qué objeto actúa? Lógicamente sobre el objeto referenciado por **this**; esto es, sobre el objeto *CArticulos* que recibió el mensaje especificado por el método.

ACCESO ALEATORIO A FICHEROS EN EL DISCO

Hasta este punto, hemos trabajado con ficheros de acuerdo con el siguiente esquema: abrir el fichero, leer o escribir hasta el final del mismo y cerrar el fichero. Pero no hemos leído o escrito a partir de una determinada posición dentro del fichero. Esto es particularmente importante cuando necesitamos modificar algunos de los valores contenidos en el fichero.

La biblioteca estándar de C++ a través de sus clases **istream** y **ostream** permiten este tipo de acceso directo. La clase **istream** provee los métodos **seekg** y **tellg**, que están definidos de la forma siguiente:

```
istream& seekg(pos_type pos)
istream& seekg(off_type des, ios_base::seekdir pos)
```

La primera versión de este método cambia la posición actual de escritura a la posición *pos* de carácter del fichero, y la segunda desplaza la posición actual de escritura *desp* caracteres respecto de la posición *pos* que puede ser:

```
ios::beg    principio del flujo.
ios::cur    posición actual en el flujo del puntero de lectura.
ios::end    final del flujo.
```

Si la operación falla, se activa el indicador **failbit**.

```
pos_type tellg()
```

Este método da como resultado la posición actual en el flujo del puntero de lectura. Esta posición es relativa al principio del búfer del flujo.

Análogamente, la clase **ostream** provee los métodos **seekp** y **tellp**, que están definidos como se indica a continuación:

```
ostream& seekp(pos_type pos)
ostream& seekp(off_type des, ios_base::seekdir pos)
pos_type tellp()
```

La explicación para estos métodos es la misma que la dada para sus homólogos de la clase **istream**.

Los sufijos *g* y *p* son necesarios porque cuando se crea un objeto de la clase **iostream**, que como sabemos está derivada de las clases **istream** y **ostream**, tal objeto necesita seguir la pista del puntero de lectura y del puntero de escritura. El sufijo *g* (*get*) indica que el puntero que hay que mover en el flujo es el de lectura, y el sufijo *p* (*put*) indica que el puntero que hay que mover es el de escritura.

Como ejemplo, vamos a modificar la aplicación anterior (fichero que almacenaba artículos de un determinado almacén) en la que intervenían las clases *CRegistro*, *CArticulos* y los ficheros *leerdatos.h* y *leerdatos.cpp*, para que permita, entre otras cosas, modificar un registro cualquiera del fichero. Esto acarrea las siguientes operaciones:

- Sobrecargaremos el operador `<<` para que permita visualizar un objeto *CRegistro*. Este método puede ser de interés para mostrar el registro que se desea actualizar.
- Modificaremos el método *CArticulo::Abrir* para que ahora presente las opciones “leer y escribir” y “añadir”.

- Añadiremos a la clase *CArticulo* el método *Modificar*.
- Modificaremos el programa *test.cpp* para añadir la opción *Modificar* en el menú que presenta.

La función **operator<<** se declarará **friend** de la clase *CRegistro*. Por lo tanto, añade la declaración de la misma a la declaración de la clase *CRegistro* y su definición en la definición de *CRegistro*, según se muestra a continuación:

```
// registro.h
class CRegistro
{
    friend std::ostream& operator<<(std::ostream&, const CRegistro&);

private:
    char referencia[30];
    float precio;
public:
    CRegistro(); // constructor
    void AsignarRegistro(char *, float);
    char *ObtenerReferencia(char *) const;
    float ObtenerPrecio() const;
};

// registro.cpp
ostream& operator<<(ostream& os, const CRegistro& reg)
{
    // Visualizar un registro
    // Salida de resultados alineados en columnas
    os << setw(32) << left // establecer ancho y ajuste a la izda.
    << setfill( '.' ) // carácter de relleno
    << reg.referencia // escribe la referencia
    << setw(10) << right // establecer ancho y ajuste a la dcha.
    << fixed << setprecision(2) // coma flotante con dos decimales
    << reg.precio << endl; // escribe precio y '\n'
    return os;
}
```

Puesto que el método *CArticulos::Modificar* necesita que el fichero esté abierto para leer y escribir, vamos a modificar el método *Abrir* de esta clase para que permita realizar esta operación:

```
void Carticulos::Abrir(string& nomf) throw(failure)
{
    // ...
    // Opciones del menú
    static char *opciones[] =
    {
```

```

"leer y escribir",
"añadir",
};
int op, nOpciones = sizeof(opciones)/sizeof(char*);

if ((op = menu(opciones, nOpciones)) == 1) // leer y escribir
    open(nomf.c_str(), ios::in | ios::out | ios::binary);
else // añadir
    open(nomf.c_str(), ios::out | ios::app | ios::binary);
modo = op;
}

```

Una operación importante en el trabajo con ficheros que se puede realizar de forma rápida y fácil cuando se permite el acceso aleatorio al mismo es modificar alguna parte concreta de la información almacenada en él. En nuestro caso, el objetivo es modificar un registro. Esta operación, si se hace partiendo de la posición que ocupa el registro en el fichero, puede que requiera buscarlo previamente. En este caso, podemos utilizar la opción *Buscar*. Bajo este planteamiento, vamos a añadir a la clase *CArticulo* un método denominado *Modificar* que realizará, en el orden descrito, básicamente las siguientes operaciones:

1. Calculará el número de registros del fichero.
2. Solicitará el número de registro que desea modificar.
3. Mostrará este registro para que el usuario pueda comprobar que se trata del registro requerido.
4. Presentará en pantalla un menú que permita al usuario modificar cualquiera de los datos del registro, guardar las modificaciones o salir sin hacer ningún cambio.
5. Lanzará una excepción de tipo **failure** si durante el proceso de E/S ocurre algún error.

```

void CArticulos::Modificar() throw(failure)
{
    // Habilitar las excepciones del tipo failure
    clear(); // desactivar los indicadores que haya activos
    exceptions(ios::failbit | ios::badbit);

    if (modo != 1)
        throw failure(string("Abrir el fichero para leer y escribir"));

    // Modificar un registro
    CRegistro reg;
    int tm = sizeof(reg); // tamaño del registro
    // Calcular el número de registros
    seekp(0L, ios::end);
}

```

```
long totalreg = tellp() / tm;

// Solicitar el número de registro a modificar
long nreg, desp;
do
{
    cout << "\nNúmero de registro entre 1 y " << totalreg
        << " (0 para salir): ";
    leerDato(nreg);
    if (nreg > 0 && nreg <= totalreg)
    {
        desp = (nreg - 1) * tm;
        seekg(desp, ios::beg); // posicionarse
        read(reinterpret_cast<char*>(&reg), tm); // leer el registro
        cout << endl << reg; // mostrar el registro leído
        // Modificar el dato referencia, precio, ambos o ninguno
        // Opciones del menú
        static char *opciones[] =
        {
            "referencia",
            "precio",
            "salir y salvar los cambios",
            "salir sin salvar los cambios"
        };
        int op, nOpciones = sizeof(opciones)/sizeof(char*);
        char ref[30];
        float pre;
        do
        {
            cout << "\nModificar el dato:";
            switch(op = menu(opciones, nOpciones))
            {
                case 1: // modificar la referencia
                    cout << "Referencia: "; leerDato(ref);
                    reg.AsignarRegistro(ref, reg.ObtenerPrecio());
                    break;
                case 2: // modificar el precio
                    cout << "Precio: "; leerDato(pre);
                    reg.AsignarRegistro(reg.ObtenerReferencia(ref), pre);
                    break;
                case 3: // salir y guardar los cambios
                    break;
                case 4: // salir sin guardar los cambios
                    break;
            }
        }
        while (op != 3 && op != 4);

        if (op == 3)
        {
            seekp(-tm, ios::cur);
        }
    }
}
```



```

        write(reinterpret_cast<char*>(&reg), tm);
    }
}
else if (nreg < 0)
    cout << "error: número de registro negativo\n";
}
while (nreg != 0);
}

```

Finalmente, modificamos el programa *test.cpp* para que incluya en el menú que presenta la opción *Modificar*:

```

int main()
{
    // Opciones del menú
    static char *opciones[] =
    {
        "Nuevo fichero",
        "Abrir fichero",
        "Añadir registros",
        "Buscar un registro",
        "Modificar",
        "Salir"
    };
    int nOpciones = sizeof(opciones)/sizeof(char*);
    // ...

    case 5: // modificar
        fs.Modificar();
        break;
    case 6: // salir
        salir = true;
    }
}
catch(ios_base::failure& e)
{
    cout << e.what() << endl;
}
if (salir) break;
}
}

```

EJERCICIOS PROPUESTOS

1. Realizar un programa que permita crear un fichero nuevo, abrir uno existente, añadir, buscar, modificar y borrar registros. El nombre del fichero será introducido a través del teclado. Cada registro del fichero será un objeto persona con los atributos *nombre*, *dirección* y *teléfono*. Así mismo, para que el usuario pueda ele-

gir cualquiera de las operaciones enunciadas, el programa visualizará en pantalla un menú similar al siguiente:

Fichero actual: ninguno

1. Nuevo fichero
 2. Abrir fichero
 3. Añadir registros
 4. Buscar un registro
 5. Buscar siguiente
 6. Modificar un registro
 7. Eliminar un registro
 8. Salir
-

Opción (1 - 8): 1

Nombre del fichero: telefonos.dat

La opción *Nuevo* abrirá un fichero para añadir registros; si el fichero existe, preguntará si se desea sobrescribir. La opción *Abrir* permitirá abrir un fichero para leer y escribir o para añadir; estas dos opciones se elegirán de un menú. La opción *Buscar* permitirá buscar un registro por el campo *nombre*; se permitirá introducir una subcadena de *nombre*, incluso vacía. La opción *Buscar siguiente* buscará el siguiente registro que cumpla las mismas condiciones que el anteriormente buscado. La opción *Modificar* permitirá cambiar el contenido de cualquier campo de un registro. Finalmente, la opción *Eliminar* permitirá marcar un registro para borrar. Se deberá realizar al menos un método para cada una de las opciones, excepto para las opciones *Buscar*, que compartirán ambas el mismo método, y para *Salir*.

A partir de un análisis del enunciado se deduce que potencialmente existen dos clases de objetos: una que represente al fichero y otra que represente a los registros del fichero.

Escribiremos entonces una clase *CPersona* para manipular cada uno de los registros de un fichero y otra *CTelefonos* con una interfaz pública que permita realizar las operaciones habituales de trabajo sobre un fichero.

Según el enunciado, la funcionalidad de la clase *CPersona* estará soportada por los atributos *nombre*, *dirección* y *teléfono* y por los métodos siguientes:

- Un constructor con parámetros por omisión para poder crear objetos *CPersona* con unos atributos determinados.
- Métodos de acceso (*asignar...* y *obtener...*) para cada uno de los atributos.

- Un método denominado *tamanyo* que devuelve la longitud en bytes correspondiente a los atributos de un objeto *CPersona*.
- Una función amiga de la clase, que sobrecargará el operador de inserción, para permitir mostrar los atributos de un objeto *CPersona*.

La funcionalidad de la clase *CTelefono* deberá permitir abrir un fichero, añadir, buscar y modificar un registro, así como leer un registro y verificar si un fichero existe. Para ello, dotaremos a esta clase del atributo *modo*, que valdrá 1 cuando el fichero haya sido abierto para “leer y escribir” o 2 cuando haya sido abierto para “añadir” registros al final del mismo, y de los siguientes métodos:

- Un constructor sin argumentos y un destructor que cierre el fichero abierto.
- *Modo*. Comprueba si el fichero fue abierto para leer y escribir o para añadir.
- *Existe*. Comprueba si un determinado fichero existe.
- *LeerReg*. Devuelve el objeto *CPersona* correspondiente al número de registro especificado.
- *Nuevo*. Permite abrir un fichero nuevo para añadir. Si el fichero especificado existe, preguntará si se desea sobrescribir.
- *Abrir*. Permite abrir un fichero para leer y escribir o para añadir.
- *Agregar*. Permite añadir uno o más registros al final del fichero.
- *Buscar*. Permite buscar un registro por una subcadena del nombre y a partir de una posición determinada dentro del fichero.
- *Modificar*. Permite modificar cualquier campo del registro especificado.
- Y *Eliminar*. Permite marcar un registro para ser eliminado.

El método *Modificar* tiene como finalidad permitir modificar cualquier campo de cualquier registro del fichero actual con el que estamos trabajando. Para ello, solicitará el número de registro a modificar, lo leerá, visualizará los campos correspondientes y presentará un menú que permita modificar cualquiera de esos campos:

Número de registro entre 1 y 53 (0 para salir): 3

Mercedes
Barcelona
93234567

Modificar el dato:

-
1. nombre
 2. dirección
 3. teléfono
 4. salir y salvar los cambios
 5. salir sin salvar los cambios
-

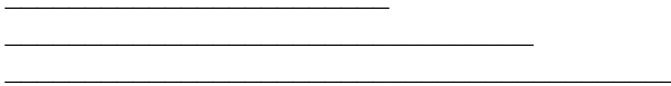
Opción (1 - 5):

Finalmente, para borrar registros del fichero, siga los pasos indicados a continuación:

- Añada a la clase *CTelefonos* un atributo *borrado*, que inicialmente valdrá **false** y tomará el valor **true** cuando se marque algún registro para borrar.
- Añada a la clase *CTelefonos* un método *TieneRegsBorrados*, que devuelva el valor del atributo *borrado*.
- Añada a la clase *CTelefonos* un método *Eliminar*, que reciba como argumento el número de teléfono cuyo registro se desea eliminar, y marque este registro poniendo en su campo *nombre* la palabra “borrar”.
- Añada a la clase *CTelefonos* un método *Actualizar* sin parámetros, que permita eliminar los registros del fichero actual marcados para borrar. Esta operación sólo será necesario hacerla si al salir de la aplicación, o al abrir un fichero diferente al actual, el atributo *borrado* vale **true**.

P A R T E

3



Diseño y programación

- Estructuras dinámicas
- Algoritmos

ESTRUCTURAS DINÁMICAS

La principal característica de las estructuras dinámicas es la facultad que tienen para variar su tamaño y hay muchos problemas que requieren de este tipo de estructuras. Esta propiedad las distingue claramente de las estructuras estáticas fundamentales como las matrices. Cuando se crea una matriz su número de elementos se fija en ese instante y después no puede agrandarse o disminuirse elemento a elemento, conservando el espacio actualmente asignado; en cambio, cuando se crea una estructura dinámica eso sí es posible.

Por tanto, no es posible asignar una cantidad fija de memoria para una estructura dinámica, y como consecuencia un compilador no puede asociar direcciones explícitas con las componentes de tales estructuras. La técnica que se utiliza más frecuentemente para resolver este problema consiste en realizar una asignación dinámica para las componentes individuales, al tiempo que son creadas durante la ejecución del programa, en vez de hacer la asignación de una sola vez para un número de componentes determinado.

Cuando se trabaja con estructuras dinámicas, el compilador asigna una cantidad fija de memoria para mantener la dirección del componente asignado dinámicamente, en vez de hacer una asignación para el componente en sí. Esto implica que debe haber una clara distinción entre datos y referencias a datos, y que consecuentemente se deben emplear tipos de datos cuyos valores sean referencias a otros datos; nos estamos refiriendo a las variables de tipo “puntero a”.

Cuando se asigna memoria dinámicamente para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Para realizar esta operación disponemos en C++ del operador **new** (véase en el capítulo 7, el apartado *Asignación dinámica de memoria*).

Este capítulo introduce técnicas en programación orientada a objetos para construir estructuras abstractas de datos. Una vez que haya trabajado los ejemplos de este capítulo, será capaz de explotar en sus aplicaciones la potencia de las listas enlazadas, pilas, colas y árboles binarios.

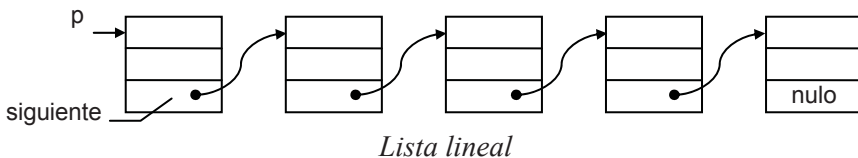
LISTAS LINEALES

Una lista lineal es una colección, originalmente vacía, de elementos u objetos de cualquier tipo no necesariamente consecutivos en memoria, que durante la ejecución del programa puede crecer o decrecer, elemento a elemento, según las necesidades previstas en el mismo. Un objeto de la clase `vector<...>` lo normal es que esté implementado como una lista lineal.

Según la definición dada surge una pregunta: si los elementos no están consecutivos en memoria, ¿cómo pasamos desde un elemento al siguiente cuando recorramos la lista? La respuesta es que cada elemento debe almacenar información de dónde está el siguiente elemento o el anterior, o bien ambos. En función de la información que cada elemento de la lista almacene respecto a la localización de sus antecesores y/o predecesores, las listas pueden clasificarse en: listas simplemente enlazadas, listas circulares, listas doblemente enlazadas y listas circulares doblemente enlazadas.

Listas lineales simplemente enlazadas

Una *lista lineal simplemente enlazada* es una colección de objetos (elementos de la lista), cada uno de los cuales contiene datos o un puntero a los datos y un puntero al siguiente objeto en la colección (elemento de la lista). Gráficamente puede representarse de la forma siguiente:



Para construir una lista lineal, primero tendremos que definir el tipo de los elementos que van a formar parte de la misma. Por ejemplo, cada elemento de la lista puede definirse como una estructura de datos con dos o más miembros: un puntero al elemento siguiente y una variable que defina el área de datos. El área de datos puede ser de un tipo predefinido o de un tipo definido por el usuario. Según esto, el tipo de cada elemento de una lista puede venir definido de la forma siguiente:


```

class CElementoLse
{
public:
    // Atributos
    // Defina aquí los datos o un puntero a los datos
    // ...
    CElementoLse *siguiente; // puntero al siguiente elemento

    // Métodos
    CElementoLse() {} // constructor sin parámetros
    // ...
};

```

Se puede observar que la clase *CElementoLse* definirá una serie de atributos correspondientes a los datos que deseemos manipular, además de un atributo especial, denominado *siguiente*, para permitir que cada elemento pueda hacer referencia a su sucesor formando así una lista enlazada.

Una vez creada la clase de objetos *CElementoLse* la asignación de memoria para un elemento se haría así:

```

int main()
{
    CElementoLse *p = 0; // puntero a un elemento
    // Asignar memoria para un elemento
    p = new CElementoLse();
    // Este elemento no tiene un sucesor
    p->siguiente = 0;
    // Operaciones cualesquiera
    // Liberar la memoria ocupada por el elemento p
    delete p;
}

```

El código *CElementoLse *p* define un puntero *p* a un objeto de la clase *CElementoLse*. La sentencia *p = new CElementoLse()* crea (asigna memoria para) un objeto de tipo *CElementoLse*, genera un puntero (dirección de memoria) que direcciona este nuevo objeto y asigna este puntero a la variable *p*. La sentencia *p->siguiente = 0* asigna al miembro *siguiente* del objeto apuntado por *p* el valor cero (puntero nulo), indicando así que después de este elemento no hay otro; esto es, que este elemento es el último de la lista.

El valor cero, puntero nulo, permite crear estructuras de datos finitas. Así mismo, suponiendo que *p* apunta al principio de la lista, diremos que dicha lista está vacía si *p* vale cero. Por ejemplo, después de ejecutar las sentencias:

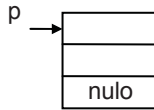
```

p = 0; // lista vacía
p = new CElementoLse(); // elemento p

```

```
p->siguiente = 0;    // no hay siguiente elemento
```

tenemos una lista de un elemento:



Para añadir un nuevo elemento a la lista, procederemos así:

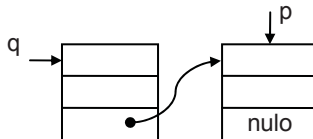
```
q = new CElementoLse(); // crear un nuevo elemento
q->siguiente = p; // almacenar la dirección del elemento siguiente
p = q; // p apunta al principio de la lista
```

donde q es un puntero a un objeto de tipo *CElementoLse*. Ahora tenemos una lista de dos elementos. Observe que los elementos nuevos se añaden al principio de la lista.

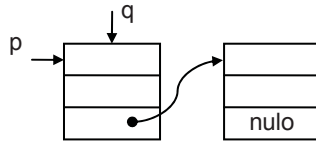
Para verlo con claridad analicemos las tres sentencias anteriores. Partimos de que tenemos una lista referenciada por p con un solo elemento. La sentencia $q = \text{new CElementoLse}()$ crea un nuevo elemento:



La sentencia $q->\text{siguiente} = p$ hace que el sucesor del elemento creado sea el anteriormente creado. Observe que ahora $q->\text{siguiente}$ y p tienen el mismo valor; esto es, la misma dirección, por lo tanto, apuntan el mismo elemento:



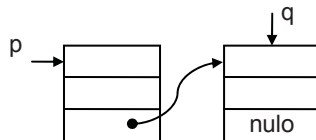
Por último, la sentencia $p = q$ hace que la lista quede de nuevo apuntada por p ; es decir, para nosotros p es siempre el primer elemento de la lista.



Ahora p y q apuntan al mismo elemento, al primero. Si ahora se ejecutara una sentencia como la siguiente, ¿qué sucedería?

```
q = q->siguiente;
```

¿Quién es q ->*siguiente*? Es el atributo *siguiente* del objeto apuntado por q que contiene la dirección de memoria donde se localiza el siguiente elemento al apuntado por p . Si este valor se lo asignamos a q , entonces q apuntará al mismo elemento que apuntaba q ->*siguiente*. El resultado es que q apunta ahora al siguiente elemento como se puede ver en la figura mostrada a continuación:



Esto nos da una idea de cómo avanzar elemento a elemento sobre una lista. Si ejecutamos de nuevo la misma sentencia:

```
q = q->siguiente;
```

¿Qué sucede? Sucede que como q ->*siguiente* vale cero, a q se le ha asignado el valor cero. Conclusión: cuando en una lista utilizamos un puntero para ir de un elemento al siguiente, en el ejemplo anterior q , diremos que hemos llegado al final de la lista cuando q tome el valor cero.

Operaciones básicas

Las operaciones que podemos realizar con listas incluyen fundamentalmente las siguientes:

1. Insertar un elemento en una lista.
2. Buscar un elemento en una lista.
3. Borrar un elemento de una lista.
4. Recorrer los elementos de una lista.
5. Borrar todos los elementos de una lista.

Partiendo de las definiciones:

```
class CElementoLse
{
public:
    // Atributos
    int dato;
    CElementoLse *siguiente; // puntero al siguiente elemento
public:
    // Métodos
    CElementoLse(int d = 0) // constructor
    {
        dato = d;
        siguiente = 0;
    }
};

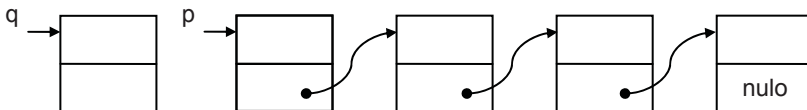
int main()
{
    CElementoLse *p = 0, *q = 0, *r = 0; // punteros
    // ...
}
```

vamos a exponer en los siguientes apartados cómo realizar cada una de las operaciones básicas. Observe que, por sencillez, vamos a trabajar con una lista de enteros.

Inserción de un elemento al comienzo de la lista

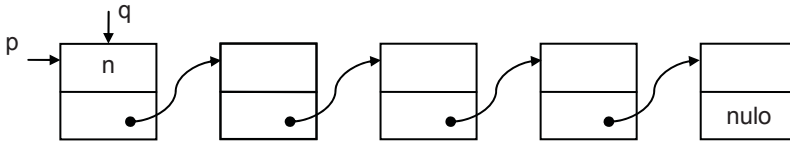
Supongamos una lista lineal referenciada por p . Para insertar un elemento al principio de la lista, primero se crea el elemento y después se reasignan las referencias, tal como se indica a continuación:

```
q = new CElementoLse();
```



```
q->dato = n; // asignación de valores
q->siguiente = p; // reasignación de punteros
p = q;
```

El orden en el que se realizan estas operaciones es esencial. El resultado es:



Esta operación básica nos sugiere cómo crear una lista. Para ello, y partiendo de una lista vacía, no tenemos más que repetir la operación de insertar un elemento al comienzo de una lista, tantas veces como elementos deseemos que tenga dicha lista. Veámoslo a continuación:

```

////////////////////////////////////
// Crear una lista lineal simplemente enlazada
//
int main()
{
    CElementoLse *p = 0, *q = 0; // punteros
    int n = 0;

    // Crear una lista de enteros
    cout << "Introducir datos. Finalizar con Ctrl+Z.\n";

    cout << "dato: ";
    while (leerDato(n))
    {
        q = new CElementoLse();
        q->dato = n;
        q->siguiente = p;
        p = q;
        cout << "dato: ";
    }
}

```

Notar que el orden de los elementos en la lista es inverso al orden en el que han llegado. Así mismo, utilizamos la plantilla *leerDato* diseñada en el capítulo 13 para leer datos desde el teclado (archivos *leerdatos.h* y *leerdatos.cpp*).

Buscar en una lista un elemento con un valor x

Supongamos que queremos buscar un determinado elemento en una lista cuyo primer elemento está apuntado por p . La búsqueda es secuencial y termina cuando se encuentra el elemento, o bien cuando se llega al final de la lista.

```

q = p; // q apunta al primer elemento de la lista
cout << "dato a buscar: "; leerDato(x);
while (q != 0 && q->dato != x)
    q = q->siguiente; // q apunta al siguiente elemento

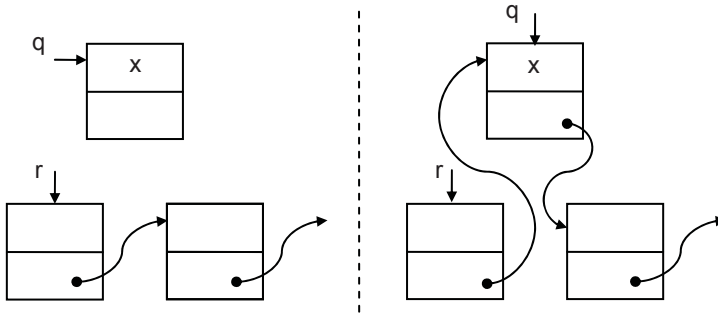
```

Observe el orden de las expresiones que forman la condición del bucle **while**. Sabemos que en una operación **&&** (AND), cuando una de las expresiones es falsa, la condición ya es falsa, por lo que el resto de las expresiones no necesitan ser evaluadas. De ahí que cuando q valga cero, la expresión $q \rightarrow \text{dato}$ no será evaluada, de lo contrario se produciría un error. Finalmente, la variable q quedará apuntando al elemento buscado, o valdrá cero si ese elemento no se encuentra.

Inserción de un elemento en general

La inserción de un elemento en la lista, a continuación de otro elemento cualquiera apuntado por r , es de la forma siguiente:

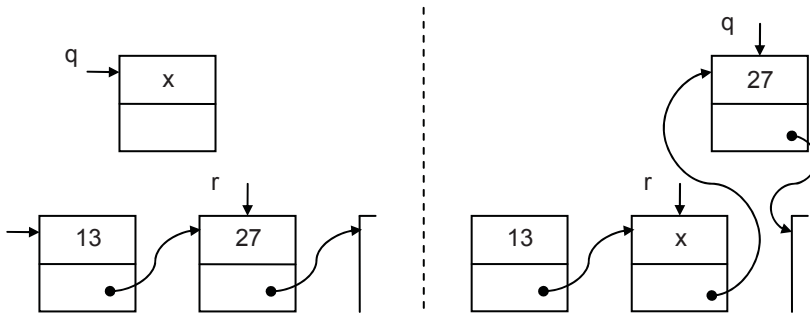
```
q = new CElementoLse();
q->dato = x; // valor insertado
q->siguiente = r->siguiente;
r->siguiente = q;
```



Inserción en la lista detrás del elemento apuntado por r

La inserción de un elemento en la lista antes de otro elemento apuntado por r se hace insertando un nuevo elemento detrás del elemento apuntado por r , intercambiando previamente los valores del nuevo elemento y del elemento apuntado por r .

```
q = new CElementoLse();
*q = *r; // copiar miembro a miembro un objeto en otro
r->dato = x; // valor insertado
r->siguiente = q;
```

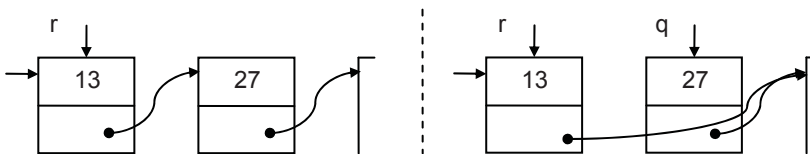


Inserción en la lista antes del elemento apuntado por r

Borrar un elemento de la lista

Para borrar el sucesor de un elemento apuntado por r , las operaciones a realizar son las siguientes:

```
q = r->siguiente;           // q apunta al elemento a borrar
r->siguiente = q->siguiente; // enlazar los elementos anterior
                             // y posterior al borrado
delete q; // borrar el elemento apuntado por q
```

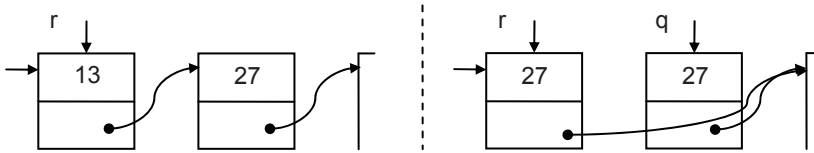


Borrar el sucesor del elemento apuntado por r

Observe que para acceder a los miembros de un elemento, éste tiene que estar referenciado por una variable. Por esta razón, lo primero que hemos hecho ha sido apuntar el elemento a borrar por q .

Para borrar un elemento apuntado por r , las operaciones a realizar son las siguientes:

```
q = r->siguiente;
*r = *q;           // copiar miembro a miembro un objeto en otro
delete q;         // borrar el objeto apuntado por q
```



Borrar el elemento apuntado por r

Como ejercicio, escribir la secuencia de operaciones que permitan borrar el último elemento de una lista.

Recorrer una lista

Supongamos que hay que realizar una operación con todos los elementos de una lista, cuyo primer elemento está apuntado por p . Por ejemplo, escribir el valor de cada elemento de la lista. La secuencia de operaciones sería la siguiente:

```
q = p; // salvar el puntero al primer elemento de la lista
while (q != 0)
{
    cout << q->dato << " ";
    q = q->siguiente;
}
```

Borrar todos los elementos de una lista

Borrar todos los elementos de una lista equivale a liberar la memoria asignada a cada uno de los elementos de la misma. Supongamos que queremos borrar una lista, cuyo primer elemento está apuntado por p . La secuencia de operaciones es la siguiente:

```
while (p != 0)
{
    q = p;
    p = p->siguiente;
    delete q;
}
```

Observe que, antes de borrar el elemento apuntado por q , hacemos que p apunte al siguiente elemento, porque si no perderíamos el resto de la lista (la apuntada por $q->siguiente$). Y, ¿por qué perderíamos la lista? Porque se pierde la única referencia que nos da acceso a la misma.

UNA CLASE PARA LISTAS LINEALES

Basándonos en las operaciones básicas sobre listas lineales descritas anteriormente, vamos a escribir a continuación una clase que permita crear una lista lineal simplemente enlazada en la que cada elemento conste de dos miembros: un valor real de tipo **double** y un puntero a un elemento del mismo tipo.

La clase la denominaremos *CListaLinealSE* (Clase *Lista Lineal Simplemente Enlazada*). Dicha clase incluirá un atributo *p* para almacenar de forma permanente una referencia al primer elemento de la lista, otro *numeroDeElementos* para almacenar el número de elementos que tiene actualmente la lista y una clase interna *CElemento* que definirá la estructura de un elemento de la lista, que según hemos indicado anteriormente será así:

```
class CElemento
{
    // Atributos
    public:
        double dato;
        CElemento *siguiente; // siguiente elemento

    // Métodos
    public:
        CElemento() { siguiente = 0; } // constructor
};
```

Finalmente, para simplificar, la interfaz pública de la clase *CListaLinealSE* proporcionará solamente los métodos siguientes: un constructor sin parámetros, un destructor, *anyadirAlPrincipio*, *tamanyo* y *obtener*.

El constructor dará lugar a una lista vacía. El destructor destruirá la lista borrando todos sus elementos. El método *anyadirAlPrincipio* permitirá añadir un nuevo elemento al principio de la lista, en nuestro caso un valor de tipo **double** recibido como parámetro por el método, *tamanyo* devolverá el número de elementos de la lista y *obtener* recibirá como parámetro la posición del elemento que se desea obtener (la primera posición es la cero) y devolverá como resultado el dato almacenado por este elemento, o bien lanzará una excepción si la lista está vacía o la posición especificada está fuera de límites.

Según lo expuesto, *CListaLinealSE* puede ser así:

```
// ListaLinealSE.h - Declaración de la clase CListaLinealSE
#ifdef !defined( _LISTALINEALSE_H_ )
#define _LISTALINEALSE_H_
```

```
////////////////////////////////////
```

```

// Lista lineal simplemente enlazada
//
class CListaLinealSE
{
private:
    // Elemento de una lista lineal simplemente enlazada
    class CElemento
    {
        // Atributos
    public:
        double dato;
        CElemento *siguiente; // siguiente elemento
        // Métodos
    public:
        CElemento() { siguiente = 0; } // constructor
    };
    // p apunta al primer elemento de la lista
    CElemento *p;
    int numeroDeElementos; // número de elementos de la lista

public:
    CListaLinealSE() { p = 0; numeroDeElementos = 0; }
    ~CListaLinealSE();
    void anyadirAlPrincipio(double n);
    int tamaño() const { return numeroDeElementos; }
    double obtener(int i) const;
};
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
#endif // _LISTALINEALSE_H_

```

```

// ListaLinealSE.cpp - Definición de la clase CListaLinealSE
#include "ListaLinealSE.h"
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Borrar los elementos de la lista
CListaLinealSE::~CListaLinealSE()
{
    CElemento *q;
    while (p != 0)
    {
        q = p;
        p = p->siguiente;
        delete q;
    }
}

// Añadir un elemento al principio de la lista
void CListaLinealSE::anyadirAlPrincipio(double n)
{
    CElemento *q = new CElemento();

```

```

    q->dato = n;        // asignación de valores
    q->siguiente = p; // reasignación de punteros
    p = q;
    numeroDeElementos++;
}

```

```
// Obtener el elemento de la posición i
```

```
double CListaLinealSE::obtener(int i) const
```

```

{
    if (p == 0) throw "lista vacía\n";

    CElemento *q = p; // apunta al primer elemento
    if (i >= 0)
    {
        // Posicionarse en el elemento i
        for (int n = 0; q != 0 && n < i; n++)
            q = q->siguiente;
        // Retornar el dato
        if (q != 0) return q->dato;
    }
    // Índice fuera de límites
    throw "índice fuera de límites\n";
}

```

```
////////////////////////////////////
```

Apoyándonos en esta clase, vamos a escribir una aplicación *Test* que cree una lista lineal simplemente enlazada que almacene una serie de valores de tipo **double** introducidos desde el teclado. Finalmente, para verificar que todo ha sucedido como esperábamos, mostraremos la lista de valores.

Para llevar a cabo lo expuesto, el método **main** de esta aplicación realizará tres cosas:

1. Definirá un objeto *lse* de la clase *CListaLinealSE*.
2. Solicitará datos de tipo **double** del teclado y los añadirá a la lista, para lo cual enviará al objeto *lse* el mensaje *anyadirAlPrincipio* por cada dato que añada.
3. Mostrará la lista de datos, para lo cual enviará al objeto *lse* el mensaje *obtener(i)* para $i = 0, 1, 2, \dots, n-1$, siendo n el tamaño de la lista.

```

// Test.cpp - Crear una lista lineal simplemente enlazada
#include <iostream>
#include "leerdatos.h" // para leerDato()
#include "ListaLinealSE.h"
using namespace std;

int main()
{
    // Crear una lista lineal vacía
    CListaLinealSE lse;

```

```

// Leer datos reales y añadirlos a la lista
double n;
cout << "Introducir datos. Finalizar con Ctrl+Z.\n";
cout << "dato: ";
while (leerDato(n))
{
    lse.anyadirAlPrincipio(n);
    cout << "dato: ";
}

// Mostrar la lista de datos
cout << endl;
double d;
int tam = lse.tamanyo();
for (int i = 0; i < tam; ++i)
{
    d = lse.obtener(i);
    cout << d << ' ';
}
}

```

Lo que hace el segmento *mostrar la lista de datos* es obtener y visualizar los valores de los elementos 0, 1, 2, ..., $tam-1$, de la lista *lse*. No obstante, es preciso señalar que ésta es una forma poco eficiente de visualizar una lista, puesto que el método *obtener* para acceder a cada elemento tiene que iniciar el avance hacia él siempre desde el principio de la lista, no desde el último elemento obtenido. La solución sería sencilla si mantenemos un puntero al último elemento accedido, cuestión que abordaremos al hablar de listas circulares doblemente enlazadas.

Clase genérica para listas lineales

La clase *CListaLinealSE* implementada anteriormente ha sido diseñada para manipular listas de un tipo específico de elementos: datos de tipo **double**. No cabe duda que esta clase tendría un mayor interés para el usuario si estuviera diseñada para permitir listas de objetos de cualquier tipo. Ésta es la dirección en la que vamos a trabajar a continuación. En otros lenguajes como C# o Java, esto se hace utilizando clases genéricas. En C++ puede hacerse utilizando plantillas.

Según esto, para que la clase *CListaLinealSE* permita listas de objetos de cualquier tipo, basta con que su clase interna *CElemento* (clase de cada uno de los elementos de la lista) tenga un atributo que sea de un tipo genérico *T*. Un tipo así definido tiene que pertenecer a una plantilla.

Esta modificación implica que el parámetro del método *anyadirAlPrincipio* tiene que ser ahora de tipo *T*, y el método *obtener* tiene que devolver ahora una

referencia a un objeto de tipo T . Todo ello lo veremos con detalle en el código que se muestra un poco más adelante.

Resumiendo, estamos hablando de una plantilla de clases *CListaLinealSE* $\langle T \rangle$. Ahora, para definir una lista *lse* de elementos de tipo **double** procederíamos así:

```
CListaLinealSE<double> lse;
```

Para completar la plantilla *CListaLinealSE* vamos a añadir otros métodos de interés. Todos se describen en la tabla siguiente:

Método	Significado
<i>CListaLinealSE</i>	Constructor sin parámetros. Construye una lista vacía.
<i>~CListaLinealSE</i>	Destructor. Borra todos los elementos de la lista.
<i>tamanyo</i>	Devuelve el número de elementos de la lista. No tiene parámetros.
<i>anyadir</i>	Añade un elemento en la posición i . Tiene dos parámetros: posición i y una referencia al objeto a añadir. Devuelve true si la operación se ejecuta satisfactoriamente y false en caso contrario.
<i>anyadirAlPrincipio</i>	Añade un elemento al principio. Tiene un parámetro que es una referencia al objeto a añadir. Devuelve true o false , igual que <i>anyadir</i> .
<i>anyadirAlFinal</i>	Añade un elemento al final. Tiene un parámetro que es una referencia al objeto a añadir. Devuelve true o false , igual que <i>anyadir</i> .
<i>borrar</i>	Borra el elemento de la posición i . Tiene un parámetro que indica la posición i del objeto a borrar. Devuelve el objeto <i>datos</i> del elemento borrado. Si la lista está vacía o el índice fuera de límites, lanza una excepción de tipo const char * .
<i>borrarPrimero</i>	Borra el primer elemento. No tiene parámetros. Por lo demás, se comporta igual que el método <i>borrar</i> .
<i>borrarUltimo</i>	Borra el último elemento. No tiene parámetros. Por lo demás, se comporta igual que el método <i>borrar</i> .
<i>obtener</i>	Devuelve el elemento de la posición i que se pasa como parámetro. Si la lista está vacía o el índice fuera de límites, lanza una excepción de tipo const char * .
<i>operador []</i>	Se comporta igual que el método <i>obtener</i> .
<i>obtenerPrimero</i>	Devuelve una referencia al objeto <i>datos</i> del primer elemento. Por lo demás, se comporta igual que el método <i>obtener</i> .
<i>obtenerUltimo</i>	Devuelve una referencia al objeto <i>datos</i> del último elemento. Por lo demás, se comporta igual que el método <i>obtener</i> .

A continuación se muestra el código completo de la plantilla *CListaLinealSE*. Observar que, además de los métodos especificados en la tabla anterior, se ha añadido a la clase *CElemento* un constructor con parámetros.

```
// ListaLinealSE.h - Declaración de la plantilla CListaLinealSE
#if !defined( _LISTALINEALSE_H_ )
#define _LISTALINEALSE_H_

////////////////////////////////////
// Lista lineal simplemente enlazada
//
template<class T>
class CListaLinealSE
{
private:
    // Elemento de una lista lineal simplemente enlazada
    class CElemento
    {
        // Atributos
    public:
        T datos;
        CElemento *siguiente; // siguiente elemento
    // Métodos
    public:
        CElemento() { siguiente = 0; } // constructor
        CElemento(T& d, CElemento *s) // constructor
        {
            datos = d;
            siguiente = s;
        }
    };
    // p apunta al primer elemento de la lista
    CElemento *p;
    int nElementos; // número de elementos de la lista

public:
    CListaLinealSE() { p = 0; nElementos = 0; }
    ~CListaLinealSE();
    int tamaño() const { return nElementos; }
    bool anyadir(int i, T& obj);
    bool anyadirAlPrincipio(T& obj);
    bool anyadirAlFinal(T& obj);
    T borrar(int i);
    T borrarPrimero() const;
    T borrarUltimo() const;
    T& obtener(int i) const;
    T& obtenerPrimero()const;
    T& obtenerUltimo()const;
    T& operator[](int i) const;
};
```

```
////////////////////////////////////  
#include "ListaLinealSE.cpp"  
  
#endif // _LISTALINEALSE_H_  
  
// ListaLinealSE.cpp - Definición de la plantilla CListaLinealSE  
#include <iostream>  
using namespace std;  
  
////////////////////////////////////  
// Borrar los elementos de la lista  
template<class T>  
CListaLinealSE<T>::~CListaLinealSE()  
{  
    CElemento *q;  
    while (p != 0)  
    {  
        q = p;  
        p = p->siguiente;  
        delete q;  
    }  
    nElementos = 0;  
}  
  
// Añadir un elemento en la posición i  
template<class T>  
bool CListaLinealSE<T>::anyadir(int i, T& obj)  
{  
    if (i > nElementos || i < 0)  
    {  
        cout << "índice fuera de límites\n";  
        return false;  
    }  
  
    // Crear el elemento a añadir  
    CElemento *q = new CElemento(obj, 0);  
  
    // Si la lista apuntada por p está vacía, añadirlo sin más  
    if (nElementos == 0)  
    {  
        // Añadir el primer elemento  
        p = q;  
        nElementos++;  
        return true;  
    }  
  
    // Si la lista no está vacía, encontrar el punto de inserción  
    CElemento *elemAnterior = p;  
    CElemento *elemActual = p;  
  
    // Posicionarse en el elemento i
```

```

    for (int n = 0; n < i; n++)
    {
        elemAnterior = elemActual;
        elemActual = elemActual->siguiente;
    }
    // Dos casos:
    // 1) Insertar al principio de la lista
    // 2) Insertar después del anterior (incluye insertar al final)
    if ( elemAnterior == elemActual ) // insertar al principio
    {
        q->siguiente = p;
        p = q; // cabecera
    }
    else // insertar después del anterior
    {
        q->siguiente = elemActual;
        elemAnterior->siguiente = q;
    }
    nElementos++;
    return true;
}

// Añadir un elemento al principio de la lista
template<class T>
bool CListaLinealSE<T>::anyadirAlPrincipio(T& obj)
{
    return anyadir(0, obj);
}

// Añadir un elemento al final de la lista
template<class T>
bool CListaLinealSE<T>::anyadirAlFinal(T& obj)
{
    return anyadir(nElementos, obj);
}

// Borrar el elemento de la posición i
template<class T>
T CListaLinealSE<T>::borrar(int i)
{
    if (i >= nElementos || i < 0) throw "índice fuera de límites\n";

    // Entrar en la lista y encontrar el índice del elemento
    CElemento *elemAnterior = p;
    CElemento *elemActual = p;
    // Posicionarse en el elemento i
    for (int n = 0; n < i; n++)
    {
        elemAnterior = elemActual;
        elemActual = elemActual->siguiente;
    }
}

```



```

// Dos casos:
// 1) Borrar el primer elemento de la lista
// 2) Borrar el siguiente a elemAnterior (elemActual)
if ( elemActual == p ) // 1)
    p = p->siguiente; // cabecera
else // 2)
    elemAnterior->siguiente = elemActual->siguiente;

T datosElemento = elemActual->datos;
delete elemActual;
nElementos--;
return datosElemento; // retornar los datos del elemento borrado
}

// Borrar el primer elemento
template<class T>
T CListaLinealSE<T>::borrarPrimero() const
{
    return borrar(0);
}

// Borrar el último elemento
template<class T>
T CListaLinealSE<T>::borrarUltimo() const
{
    return borrar(nElementos - 1);
}

// Obtener el elemento de la posición i
template<class T>
T& CListaLinealSE<T>::obtener(int i) const
{
    if (p == 0) throw "lista vacía\n";
    CElemento *q = p; // apunta al primer elemento
    if (i >= 0)
    {
        // Posicionarse en el elemento i
        for (int n = 0; q != 0 && n < i; n++)
            q = q->siguiente;
        // Retornar el dato
        if (q != 0) return q->datos;
    }

    // Índice fuera de límites
    throw "índice fuera de límites\n";
}

// Operador de indexación
template<class T>
T& CListaLinealSE<T>::operator[](int i) const
{

```

```

    return obtener(i);
}

// Retornar el primer elemento
template<class T>
T& CListaLinealSE<T>::obtenerPrimero() const
{
    return obtener(0);
}

// Retornar el último elemento
template<class T>
T& CListaLinealSE<T>::obtenerUltimo() const
{
    return obtener(nElementos - 1);
}
/////////////////////////////////////////////////////////////////

```

Como ejercicio, supongamos que deseamos crear una lista lineal simplemente enlazada con la intención de almacenar los nombres de los alumnos de un determinado curso y sus notas de la asignatura de Programación. Según este enunciado, ¿a qué tipo de objeto hará referencia cada elemento de la lista? Pues, a objetos cuya estructura interna sea capaz de almacenar un nombre (dato de tipo **string**) y una nota (dato de tipo **double**). Además, estos objetos podrán recibir una serie de mensajes con la intención de extraer o modificar su contenido. La clase representativa de los objetos descritos la vamos a denominar *CDatos* y puede escribirse de la forma siguiente:

```

// datos.h - Declaración de la clase CDatos
#ifndef _DATOS_H_
#define _DATOS_H_
#include <string>

class CDatos
{
private:
    std::string nombre;
    double nota;
public:
    CDatos(std::string nom = "", double n = 0); // constructor
    void asignarNombre(std::string nom);
    std::string obtenerNombre() const;
    void asignarNota(double n);
    double obtenerNota() const;
};
#endif // _DATOS_H_

```

```

// datos.cpp - Definición de la clase CDatos
#include "datos.h"

```

```

#include <string>
using namespace std;

CDatos::CDatos(string nom, double n) // constructor
{
    nombre = nom;
    nota = n;
}

void CDatos::asignarNombre(string nom)
{
    nombre = nom;
}

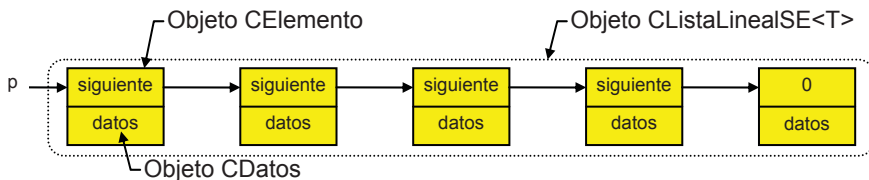
string CDatos::obtenerNombre() const
{
    return nombre;
}

void CDatos::asignarNota(double n)
{
    nota = n;
}

double CDatos::obtenerNota() const
{
    return nota;
}

```

Sólo nos queda realizar una aplicación que utilizando las clases *CListaLinealSE<T>* y *CDatos* cree una lista lineal y ponga en práctica las distintas operaciones que sobre ella pueden realizarse. La figura siguiente muestra de forma gráfica la estructura de datos que queremos construir. Observe que, en realidad, la lista lo que mantiene son referencias a los datos (objetos *CDatos*) y no los datos en sí, aunque, por sencillez, también resulta aceptable pensar que éstos forman parte de la lista lineal. La variable *p* es un puntero al elemento de índice 0; este elemento mantiene un puntero al elemento de la lista de índice 1 y un puntero al objeto de datos correspondiente, y así sucesivamente.



El código de la aplicación *Test* que se muestra a continuación enseña cómo crear y manipular una estructura de datos como la de la figura anterior:

```
// Test.cpp - Crear una lista lineal simplemente enlazada
#include <iostream>
#include "leerdatos.h" // para leerDato()
#include "ListaLinealSE.h"
#include "datos.h"
using namespace std;

void mostrarLista(const CListaLinealSE<CDatos>& lse)
{
    // Mostrar todos los elementos de la lista
    int i = 0 , tam = lse.tamano();
    while (i < tam)
    {
        cout << i << ".- " << lse[i].obtenerNombre() << ' '
             << lse[i].obtenerNota() << '\n';
        i++;
    }
}

int main()
{
    // Crear una lista lineal vacía
    CListaLinealSE<CDatos> lse;

    // Leer datos y añadirlos a la lista
    string nombre;
    double nota;
    CDatos datos;
    cout << "Introducir datos. Finalizar con Ctrl+Z.\n";
    cout << "nombre: ";
    while (leerDato(nombre))
    {
        cout << "nota:  ";
        leerDato(nota);
        datos = CDatos(nombre, nota);
        lse.anyadirAlFinal(datos);
        cout << "nombre: ";
    }

    // Añadir un objeto al principio
    datos = CDatos("abcd", 10);
    lse.anyadirAlPrincipio(datos);
    // Añadir un objeto en la posición 1
    datos = CDatos("defg", 9.5);
    lse.anyadir(1, datos);

    cout << "\n\n";
    // Mostrar el primero
    CDatos obj = lse.obtenerPrimero();
    cout << "Primero: " << obj.obtenerNombre() << ' '
         << obj.obtenerNota() << '\n';
}
```

```

// Mostrar el último
obj = lse.obtenerUltimo();
cout << "Último: " << obj.obtenerNombre() << ' '
    << obj.obtenerNota() << '\n';
// Mostrar todos
cout << "Lista:\n";
mostrarLista(lse);

// Borrar el elemento de índice 2
obj = lse.borrar(2);

// Modificar el elemento de índice 1
CDatos& robj = lse.obtener(1);
robj.asignarNota(9);

// Mostrar todos
cout << "Lista:\n";
mostrarLista(lse);
}

```

Consistencia de la aplicación

Evidentemente, los ejemplos escritos a lo largo de esta obra tienen como objetivo facilitar el aprendizaje de los temas expuestos y proporcionar una base a partir de la cual el lector pueda construir aplicaciones profesionales. Quiere esto decir que para el autor prima el aspecto didáctico, dejando para el lector el refinamiento de las aplicaciones en la medida que los conocimientos adquiridos se lo permitan. Por ejemplo, si observamos la función *mostrarLista* de la aplicación *Test.cpp* anterior, tiene un parámetro *lse* que hace referencia a un objeto **const**:

```

void mostrarLista(const CListaLinealSE<CDatos>& lse)
{
    // Mostrar todos los elementos de la lista
    int i = 0 , tam = lse.tamano();
    while (i < tam)
    {
        cout << i << ".- " << lse[i].obtenerNombre() << ' '
            << lse[i].obtenerNota() << '\n';
        i++;
    }
}

```

El hecho de que *lse* sea **const** implica que los métodos que operen sobre él tengan que ser calificados **const**; por ejemplo, *tamano*. Esto, en principio, no altera la forma en la que puede utilizarse la clase, porque si un usuario decide declarar *lse* no constante todo seguiría funcionando normalmente, ya que un método **const** puede ser invocado por un objeto constante o no constante.

Ahora bien, ¿por qué un usuario puede querer que *lse* sea una referencia a un objeto **const**? Como sabemos, una referencia se utiliza por eficiencia y el **const** por seguridad (véase *Referencias* en el capítulo 7 y *Métodos y objetos constantes* en el capítulo 9). Por ejemplo, si *mostrarLista* intentará ejecutar el código siguiente, ¿mostraría el compilador un error indicando que el objeto *lse* es constante y, por lo tanto, no se puede modificar?

```
CDatos obj("", 0.0);
lse[i] = obj; // modificar el elemento lse[i] con el valor de obj
```

En este caso no mostraría un error porque no se está modificando *lse*; el objeto *lse* encapsula sólo el puntero que apunta al principio de la lista; esto es, los elementos de la lista (*lse[i]*) no forman parte del objeto. Pero podríamos conseguir ese comportamiento escribiendo las dos versiones siguientes del operador de indexación: una para objetos no **const** y otra para objetos **const**.

```
template<class T>
T& CListaLinealSE<T>::operator[](int i)
{
    return obtener(i);
}

template<class T>
const T& CListaLinealSE<T>::operator[](int i) const
{
    return const_cast<CListaLinealSE<T> *>(this)->operator[](i);
}
```

La primera versión, por ser un método no **const**, sería invocada para objetos no constantes de su clase. Como devuelve una referencia a un objeto de tipo *T*, permitiría también modificar ese objeto. Por ejemplo, la segunda sentencia de la función *iniciarElemento* que se muestra a continuación invoca a esta primera versión, ya que *lse* es un objeto no **const**:

```
void iniciarElemento(CListaLinealSE<CDatos>& lse, int i)
{
    CDatos obj("", 0.0);
    lse[i] = obj; // invoca a operator[](int i)
}
```

La segunda versión (sobrecarga de la primera), por ser un método **const**, sólo sería invocada para objetos constantes de su clase. Como devuelve una referencia a un objeto **const** de tipo *T*, no permitiría modificar ese objeto. Por ejemplo, el código de la función *mostrarElemento* que se expone a continuación invoca a esta segunda versión, ya que *lse* hace referencia a un objeto **const**:

```

void mostrarElemento(const CListaLinealSE<CDatos>& lse, int i)
{
    cout << i << ".- " << lse[i].obtenerNombre() << ' '
         << lse[i].obtenerNota() << '\n';
    // lse[i] invoca a operator[](int i) const
}

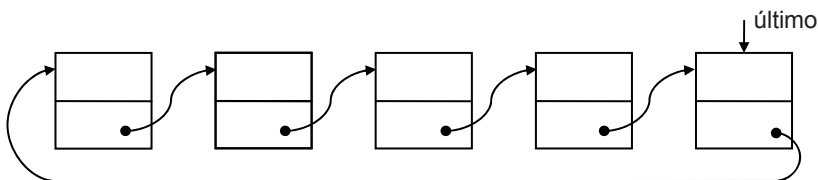
```

En este mismo sentido habría que modificar otros métodos implicados como *obtener*, *obtenerPrimero* y *obtenerUltimo*.

Como se puede observar, el simple hecho de que un intento de modificar una lista por parte de una función como *mostrarElemento* sea detectado durante la compilación, nos ha llevado a una exposición que se aparta del tema que estábamos explicando: lista lineal simplemente enlazada. Por eso en lo sucesivo, estos detalles de implementación los dejaremos como ejercicio para el lector.

LISTAS CIRCULARES

Una *lista circular* es una lista lineal en la que el último elemento apunta al primero. Entonces es posible acceder a cualquier elemento de la lista desde cualquier punto dado. Las operaciones sobre una lista circular resultan más sencillas, ya que se evitan casos especiales. Por ejemplo, el método *anyadir* de la clase *CListaLinealSE<T>* expuesta anteriormente contempla dos casos: insertar al principio de la lista e insertar a continuación de un elemento. Con una lista circular, estos dos casos se reducen a uno. La siguiente figura muestra cómo se ve una lista circular simplemente enlazada.



Cuando recorremos una lista circular, diremos que hemos llegado al final de la misma cuando nos encontremos de nuevo en el punto de partida, suponiendo, desde luego, que el punto de partida se guarda de alguna manera en la lista; por ejemplo, con un puntero fijo al mismo. Este puntero puede ser al primer elemento de la lista; también puede ser al último elemento, en cuyo caso también es conocida la dirección del primer elemento. Otra posible solución sería poner un elemento especial identificable en cada lista circular como lugar de partida. Este elemento especial recibe el nombre de elemento de *cabecera* de la lista. Esto presenta, además, la ventaja de que la lista circular no estará nunca vacía.

Como ejemplo, vamos a construir una lista circular con una referencia fija al último elemento. Una lista circular con una referencia al último elemento es equivalente a una lista lineal recta con dos referencias, una al principio y otra al final.

Para construir una lista circular, primero tendremos que definir la clase de objetos que van a formar parte de la misma. Por ejemplo, cada elemento de la lista puede definirse como una estructura de datos con dos miembros: un puntero al elemento siguiente y una variable que defina el área de datos. El área de datos puede ser de un tipo predefinido o de un tipo definido por el usuario. Según esto, el tipo de cada elemento de la lista puede venir definido de la forma siguiente:

```
class CElemento
{
    // Atributos
public:
    T datos;
    CElemento *siguiente; // siguiente elemento
    // Métodos
public:
    CElemento() { siguiente = 0; } // constructor
    CElemento(T& d, CElemento *s) // constructor
    {
        datos = d;
        siguiente = s;
    }
};
```

Vemos que por tratarse de una lista lineal simplemente enlazada, aunque sea circular, la estructura de sus elementos no varía con respecto a lo estudiado anteriormente.

Podemos automatizar el proceso de implementar una lista circular diseñando una clase *CListaCircularSE<T>* (Clase *Lista Circular Simplemente Enlazada*) que proporcione los atributos y métodos necesarios para crear cada elemento de la lista, así como para permitir el acceso a los mismos. Esta clase nos permitirá posteriormente derivar otras clases que sean más específicas; por ejemplo, para manipular *pilas* o para manipular *colas*. Estas estructuras de datos las estudiaremos un poco más adelante.

Clase CListaCircularSE<T>

La clase *CListaCircularSE<T>* que vamos a implementar incluirá un atributo *ultimo* que valdrá cero cuando la lista esté vacía y cuando no, apuntará siempre a su último elemento y otro, *nElementos*, que almacene el número actual de elementos de la lista; también incluirá una clase interna, *CElemento*, que definirá la estructura de los elementos, así como los métodos indicados en la tabla siguiente:

Método	Significado
<i>CListaCircularSE</i>	Constructor sin parámetros. Construye una lista vacía.
<i>~CListaCircularSE</i>	Destructor. Borra todos los elementos de la lista.
<i>tamanyo</i>	Devuelve el número de elementos de la lista. No tiene parámetros.
<i>anyadirAlPrincipio</i>	Añade un elemento al principio (el primer elemento es el referenciado por <i>ultimo->siguiente</i>). Tiene un parámetro que es una referencia al objeto de tipo <i>T</i> a añadir. No devuelve ningún valor.
<i>anyadirAlFinal</i>	Añade un elemento al final (el último elemento siempre estará referenciado por <i>ultimo</i>). Tiene un parámetro que es una referencia al objeto de tipo <i>T</i> a añadir. No devuelve ningún valor.
<i>borrar</i>	Borra el elemento primero (el primer elemento es el referenciado por <i>ultimo->siguiente</i>). No tiene parámetros. Devuelve los datos del elemento borrado. Si la lista está vacía, lanza una excepción de tipo const char * .
<i>obtener</i>	Devuelve los datos del elemento de la posición <i>i</i> que se pasa como parámetro. Si la lista está vacía o el índice fuera de límites, lanza una excepción de tipo const char * .
<i>operador []</i>	Se comporta igual que el método <i>obtener</i> .

A continuación se presenta el código correspondiente a la definición de la plantilla *CListaCircularSE*:

```
// ListaCircularSE.h - Declaración de la plantilla CListaCircularSE
#if !defined( _LISTCIRCULARSE_H_ )
#define _LISTCIRCULARSE_H_

////////////////////////////////////
// Lista circular simplemente enlazada
//
template<class T>
class CListaCircularSE
{
private:
    // Elemento de una lista circular simplemente enlazada
    class CElemento
    {
    // Atributos
    public:
        T datos;
        CElemento *siguiente; // siguiente elemento
    // Métodos
    public:
        CElemento() { siguiente = 0; } // constructor
    };
};
```

```

        CElemento(T& d, CElemento *s) // constructor
        {
            datos = d;
            siguiente = s;
        }
};
// ultimo: apunta al último elemento.
// ultimo->siguiente apunta al primer elemento de la lista.
CElemento *ultimo;
int nElementos; // número de elementos de la lista

public:
    CListaCircularSE() { ultimo = 0; nElementos = 0; }
    ~CListaCircularSE();
    int tamaño() const { return nElementos; };
    void anyadirAlPrincipio(T& obj);
    void anyadirAlFinal(T& obj);
    T borrar();
    T& obtener(int i) const;
    T& operator[](int i) const;
};
////////////////////////////////////
#include "ListaCircularSE.cpp"
////////////////////////////////////

#endif // _LISTCIRCULARSE_H_

// ListaCircularSE.cpp -Definición de la plantilla CListaCircularSE
////////////////////////////////////
// Borrar los elementos de la lista
template<class T>
CListaCircularSE<T>::~~CListaCircularSE()
{
    if (ultimo == 0) return;
    CElemento *q = ultimo->siguiente; // primer elemento
    while (q != ultimo)
    {
        ultimo->siguiente = q->siguiente;
        delete q;
        q = ultimo->siguiente;
    }
    delete ultimo;
    nElementos = 0;
}

// Añadir un elemento al principio de la lista
template<class T>
void CListaCircularSE<T>::anyadirAlPrincipio(T& obj)
{
    // Crear el nuevo elemento.
    CElemento *q = new CElemento(obj, 0);

```

```

    if( ultimo != 0 ) // existe una lista
    {
        q->siguiente = ultimo->siguiente;
        ultimo->siguiente = q;
    }
    else // inserción del primer elemento
    {
        ultimo = q;
        ultimo->siguiente = q;
    }
    nElementos++;
}

// Añadir un elemento al final de la lista
template<class T>
void CListaCircularSE<T>::anyadirAlFinal(T& obj)
{
    // Crear el nuevo elemento.
    // ultimo apuntará a este nuevo elemento.
    CElemento *q = new CElemento(obj, 0);

    if( ultimo != 0 ) // existe una lista
    {
        q->siguiente = ultimo->siguiente;
        ultimo = ultimo->siguiente = q;
    }
    else // inserción del primer elemento
    {
        ultimo = q;
        ultimo->siguiente = q;
    }
    nElementos++;
}

// Devolver los datos del primer elemento de
// la lista y borrar este elemento.
template<class T>
T CListaCircularSE<T>::borrar()
{
    if( ultimo == 0 ) throw "Lista vacía\n";
    CElemento *q = ultimo->siguiente;
    T obj = q->datos;
    if( q == ultimo )
        ultimo = 0;
    else
        ultimo->siguiente = q->siguiente;
    delete q;
    nElementos--;
    return obj;
}

```

```

// Obtener el elemento de la posición i
template<class T>
T& CListaCircularSE<T>::obtener(int i) const
{
    if( ultimo == 0 ) throw "Lista vacía\n";
    if ( i >= nElementos || i < 0) throw "índice fuera de límites\n";

    CElemento *q = ultimo->siguiente; // primer elemento
    // Posicionarse en el elemento i
    for (int n = 0; n < i; n++)
        q = q->siguiente;
    // Retornar los datos
    return q->datos;
}

// Operador de indexación
template<class T>
T& CListaCircularSE<T>::operator[](int i) const
{
    return obtener(i);
}
/////////////////////////////////////////////////////////////////

```

Una vez que hemos escrito la plantilla *CListaCircularSE* vamos a realizar una aplicación que permita crear una lista circular y ponga a prueba las distintas operaciones que sobre ella pueden realizarse. Los elementos de esta lista serán objetos de la clase *CDatos* utilizada en ejemplos anteriores. El código de esta aplicación puede ser el siguiente:

```

// Test.cpp - Crear una lista circular simplemente enlazada
#include <iostream>
#include "leerdatos.h" // para leerDato()
#include "ListaCircularSE.h"
#include "datos.h"

using namespace std;

void mostrarLista(const CListaCircularSE<CDatos>& lcse)
{
    // Mostrar todos los elementos de la lista
    int i = 0 , tam = lcse.tamanyo();
    while (i < tam)
    {
        cout << i << ".- " << lcse[i].obtenerNombre() << ' '
            << lcse[i].obtenerNota() << '\n';
        i++;
    }
    if (tam == 0) cout << "lista vacía\n";
}

```

```
int main()
{
    // Crear una lista lineal vacía
    CListaCircularSE<CDatos> lcse;

    // Leer datos y añadirlos a la lista
    string nombre;
    double nota;
    CDatos datos;

    cout << "Introducir datos. Finalizar con Ctrl+Z.\n";
    cout << "nombre: ";
    while (leerDato(nombre))
    {
        cout << "nota:   ";
        leerDato(nota);
        datos = CDatos(nombre, nota);
        lcse.anyadirAlFinal(datos);
        cout << "nombre: ";
    }

    // Añadir un objeto al principio
    datos = CDatos("abcd", 10);
    lcse.anyadirAlPrincipio(datos);

    cout << "\n\n";
    // Mostrar la lista
    cout << "Lista:\n";
    mostrarLista(lcse);

    // Borrar el elemento primero
    CDatos obj = lcse.borrar();

    // Mostrar todos
    cout << "Lista:\n";
    mostrarLista(lcse);
}
```

PILAS

Una *pila* es una lista lineal en la que todas las inserciones y supresiones se hacen en un extremo de la lista. Un ejemplo de esta estructura es una pila de platos. En ella, el añadir o quitar platos se hace siempre por la parte superior de la pila. Este tipo de listas recibe también el nombre de listas *LIFO* (*last in first out* - último en entrar, primero en salir).

Las operaciones de meter y sacar en una pila son conocidas en los lenguajes ensambladores como *push* y *pop*, respectivamente. La operación de sacar un elemento de la pila suprime dicho elemento de la misma.

Para trabajar con pilas podemos diseñar una clase *CPila<T>* (Clase *Pila*), derivada de la clase base *CListaCircularSE<T>*, que soporte los métodos:

Método	Significado
<i>meterEnPila</i>	Mete un elemento en la cima de la pila (todas las inserciones se hacen por el principio de la lista). Tiene un parámetro que es una referencia al objeto de tipo <i>T</i> a añadir. No devuelve ningún valor.
<i>sacarDePila</i>	Saca el primer elemento de la cima de la pila, eliminándolo de la misma (todas las supresiones se hacen por el principio de la lista). No tiene parámetros. Devuelve los datos del elemento borrado, o bien lanza una excepción de tipo const char * si la pila está vacía.
<i>tamanyo</i>	Devuelve el número de elementos de la pila. No tiene parámetros.

Según lo expuesto, la definición de esta clase puede ser así:

```
// cpila.h - Declaración de la plantilla CPila
#ifndef _CPILA_H_
#define _CPILA_H_
#include "ListaCircularSE.h"

////////////////////////////////////
// Pila: lista en la que todas las inserciones y supresiones se
// hacen en un extremo de la misma.
//
template<class T>
class CPila : protected CListaCircularSE<T>
{
public:
    void meterEnPila(T& obj);
    T sacarDePila();
    int tamanyo() const;
};
////////////////////////////////////
#include "cpila.cpp"

#endif // _CPILA_H_

// cpila.cpp -Definición de la plantilla CPila
////////////////////////////////////
```

```

template<class T>
void CPila<T>::meterEnPila(T& obj)
{
    anyadirAlPrincipio(obj);
}

template<class T>
T CPila<T>::sacarDePila()
{
    return CListaCircularSE<T>::borrar();
}

template<class T>
int CPila<T>::tamanyo() const
{
    return CListaCircularSE<T>::tamanyo();
}
/////////////////////////////////////////////////////////////////

```

El constructor de la clase *CPila<T>* llama primero al constructor de la clase base que crea una lista con cero elementos. El que la lista sea circular es transparente al usuario de la clase.

Para meter el elemento referenciado por el parámetro *obj* en la pila, el método *meterEnPila* invoca al método *anyadirAlPrincipio* de la clase base *CListaCircularSE<T>*; y para sacar el elemento de la cima de la pila y eliminarlo de la misma, el método *sacarDePila* invoca al método *borrar* de la clase base.

Observe que la derivación de la clase *CPila<T>* de *CListaCircularSE<T>* oculta al usuario la interfaz pública de ésta, ya que la clase base se ha declarado protegida. Veremos una pequeña aplicación de cómo utilizar esta clase después de exponer el apartado relativo a *colas*.

COLAS

Una *cola* es una lista lineal en la que todas las inserciones se hacen por un extremo de la lista (por el final) y todas las supresiones se hacen por el otro extremo (por el principio). Por ejemplo, una fila en un banco. Este tipo de listas recibe también el nombre de listas *FIFO* (*first in first out* - primero en entrar, primero en salir). Este orden es la única forma de insertar y recuperar un elemento de la cola. Una cola no permite acceso aleatorio a un elemento específico. Tenga en cuenta que la operación de sacar elimina el elemento de la cola.

Para trabajar con colas podemos diseñar una clase *CCola<T>* (Clase *Cola*), derivada de la clase base *CListaCircularSE<T>*, que soporte los métodos:

Método	Significado
<i>meterEnCola</i>	Mete un elemento al final de la cola (todas las inserciones se hacen por el final de la lista). Tiene un parámetro que es una referencia al objeto de tipo <i>T</i> a añadir. No devuelve ningún valor.
<i>sacarDeCola</i>	Saca el primer elemento de la cola, eliminándolo de la misma (todas las supresiones se hacen por el principio de la lista). No tiene parámetros. Devuelve los datos del elemento borrado, o bien lanza una excepción de tipo const char * si la lista está vacía.
<i>tamanyo</i>	Devuelve el número de elementos de la cola. No tiene parámetros.

Según lo expuesto, la definición de esta clase puede ser así:

```
// ccola.h - Declaración de la plantilla CCola
#ifndef _CCOLA_H_
#define _CCOLA_H_
#include "ListaCircularSE.h"

////////////////////////////////////
// Cola: lista en la que todas las inserciones se hacen por un
// extremo de la lista (por el final) y todas las supresiones se
// hacen por el otro extremo (por el principio).
//
template<class T>
class CCola : protected CListaCircularSE<T>
{
public:
    void meterEnCola(T& obj);
    T sacarDeCola();
    int tamanyo() const;
};
////////////////////////////////////
#include "ccola.cpp"
#endif // _CCOLA_H_

// ccola.cpp - Definición de la plantilla CCola
////////////////////////////////////
template<class T>
void CCola<T>::meterEnCola(T& obj)
{
    anyadirAlFinal(obj);
}

template<class T>
T CCola<T>::sacarDeCola()
```



```
void mostrarPila(CPila<CDatos>& pila)
{
    // Mostrar todos los elementos de la pila
    int i = 0, tam = pila.tamanyo();
    CDatos obj;
    while (i < tam)
    {
        obj = pila.sacarDePila();
        cout << i << ".- " << obj.obtenerNombre()
            << " " << obj.obtenerNota() << '\n';
        i++;
    }
    if (tam == 0) cout << "pila vacía\n";
}

void mostrarCola(CCola<CDatos>& cola)
{
    // Mostrar todos los elementos de la cola
    int i = 0, tam = cola.tamanyo();
    CDatos obj;
    while (i < tam)
    {
        obj = cola.sacarDeCola();
        cout << i << ".- " << obj.obtenerNombre()
            << " " << obj.obtenerNota() << '\n';
        i++;
    }
    if (tam == 0) cout << "cola vacía\n";
}

int main()
{
    // Crear una pila y una cola vacías
    CPila<CDatos> pila;
    CCola<CDatos> cola;

    // Leer datos y añadirlos a ambas
    string nombre;
    double nota;
    CDatos datos;
    cout << "Introducir datos. Finalizar con Ctrl+Z.\n";
    cout << "nombre: ";
    while (leerDato(nombre))
    {
        cout << "nota:  ";
        leerDato(nota);
        datos = CDatos(nombre, nota);
        pila.meterEnPila(datos);
        cola.meterEnCola(datos);
        cout << "nombre: ";
    }
}
```

```

cout << "\n\n";

// Mostrar la pila
cout << "\nPila:\n";
mostrarPila(pila);
// Mostrar la pila por segunda vez
cout << "\nPila:\n";
mostrarPila(pila);

// Mostrar la cola
cout << "\nCola:\n";
mostrarCola cola);
// Mostrar la cola por segunda vez
cout << "\nCola:\n";
mostrarCola cola);
}

```

Si ejecutamos esta aplicación e introducimos los siguientes datos,

Introducir datos. Finalizar con Ctrl+Z.

```

nombre: Alumno 1
nota: 7.5
nombre: Alumno 2
nota: 8.5
nombre: Alumno 3
nota: 9.5
nombre: [Ctrl+Z]

```

se mostrarán los siguientes resultados, los cuales indican que las operaciones de sacar en las pilas y colas eliminan el objeto sacado de las mismas.

```

Pila:
0.- Alumno 3 9.5
1.- Alumno 2 8.5
2.- Alumno 1 7.5

```

```

Pila:
pila vacía

```

```

Cola:
0.- Alumno 1 7.5
1.- Alumno 2 8.5
2.- Alumno 3 9.5

```

```

Cola:
cola vacía

```

En estos resultados también se puede observar que en las pilas el último objeto en entrar es el primero en salir, y en las colas el primero en entrar es el primero en salir.

LISTA DOBLEMENTE ENLAZADA

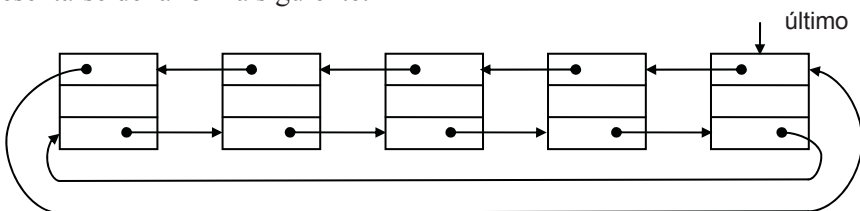
En una lista doblemente enlazada, a diferencia de una lista simplemente enlazada, cada elemento tiene información de dónde se encuentra el elemento posterior y el elemento anterior. Esto permite leer la lista en ambas direcciones. Este tipo de listas es útil cuando la inserción, borrado y movimiento de los elementos son operaciones frecuentes. Una aplicación típica es un procesador de textos, donde el acceso a cada línea individual se hace a través de una lista doblemente enlazada.

Las operaciones sobre una lista doblemente enlazada normalmente se realizan sin ninguna dificultad. Sin embargo, casi siempre es mucho más fácil la manipulación de las mismas cuando existe un doble enlace entre el último elemento y el primero, estructura que recibe el nombre de *lista circular doblemente enlazada*. Para moverse sobre una lista circular, es necesario almacenar de alguna manera un punto de referencia; por ejemplo, mediante un puntero al último elemento de la lista.

En el apartado siguiente se expone la forma de construir y manipular una lista circular doblemente enlazada.

Lista circular doblemente enlazada

Una *lista circular doblemente enlazada (lcde)* es una colección de objetos, cada uno de los cuales contiene datos o un puntero a los datos, un puntero al elemento siguiente en la colección y un puntero al elemento anterior. Gráficamente puede representarse de la forma siguiente:



Para construir una lista de este tipo, primero tendremos que definir la clase de objetos que van a formar parte de la misma. Por ejemplo, cada elemento de la lista puede definirse como una estructura de datos con tres miembros: un puntero al elemento siguiente, otro al elemento anterior y una variable que defina el área de datos. El área de datos puede ser de un tipo predefinido o de un tipo definido por

el usuario. Según esto, el tipo de cada elemento de la lista puede venir definido de la forma siguiente:

```
class CElemento
{
    // Atributos
public:
    T datos;
    CElemento *anterior; // anterior elemento
    CElemento *siguiente; // siguiente elemento
    // Métodos
public:
    CElemento() { anterior = siguiente = 0; } // constructor
    CElemento(T& d, CElemento *s) // constructor
    {
        datos = d;
        anterior = siguiente = s;
    }
};
```

Podemos automatizar el proceso de implementar una lista circular doblemente enlazada diseñando una clase *CListaCircularDE<T>* (Clase *Lista Circular Doblemente Enlazada*) que proporcione los atributos y métodos necesarios para crear cada elemento de la lista, así como para permitir el acceso a los mismos. La clase que escribimos a continuación cubre estos objetivos.

Clase *CListaCircularDE<T>*

La clase *CListaCircularDE<T>* que vamos a implementar incluirá los atributos *ultimo*, *actual*, *numeroDeElementos* y *posicion*. El atributo *ultimo* valdrá cero cuando la lista esté vacía y cuando no, apuntará siempre a su último elemento; *actual* apunta siempre al último elemento accedido; *numeroDeElementos* es el número de elementos que tiene la lista y *posicion* indica el índice del elemento referenciado por *actual*. Así mismo, incluye una clase interna, *CElemento*, que definirá la estructura de los elementos, y los métodos indicados en la tabla siguiente:

Método	Significado
<i>CListaCircularDE</i>	Es el constructor. Inicia <i>ultimo</i> y <i>actual</i> a 0 (puntero nulo), <i>numeroDeElementos</i> a 0 y <i>posicion</i> a -1 (la posición del primer elemento de la lista es la 0).
<i>~CListaCircularDE</i>	Es el destructor. Borra todos los elementos de la lista.
<i>tamanyo</i>	Devuelve el número de elementos de la lista. No tiene parámetros.

Método	Significado
<i>insertar</i>	Añade un elemento a continuación del referenciado por <i>actual</i> . El elemento añadido pasa a ser el elemento <i>actual</i> . Tiene un parámetro que es una referencia al objeto de tipo <i>T</i> a añadir. No devuelve ningún valor.
<i>borrar</i>	Borra el elemento referenciado por <i>actual</i> . No tiene parámetros. Devuelve los datos del elemento borrado o lanza una excepción de tipo const char * si la lista está vacía.
<i>irAlSiguiente</i>	Avanza la posición <i>actual</i> al siguiente elemento. Si esta posición coincide con <i>numeroDeElementos-1</i> , permanece en ella. No tiene parámetros y no devuelve ningún valor.
<i>irAlAnterior</i>	Retrasa la posición <i>actual</i> al elemento anterior. Si esta posición coincide con la <i>0</i> , permanece en ella. No tiene parámetros y no devuelve ningún valor.
<i>irAlPrincipio</i>	Hace que la posición <i>actual</i> sea la <i>0</i> . No tiene parámetros y no devuelve ningún valor.
<i>irAlFinal</i>	Hace que la posición <i>actual</i> sea la <i>numeroDeElementos-1</i> . No tiene parámetros y no devuelve ningún valor.
<i>irAl</i>	Avanza la posición <i>actual</i> al elemento de índice <i>i</i> (el primer elemento tiene índice 0). Tiene un parámetro y devuelve true si la operación de mover se realiza con éxito o false si la lista está vacía o el índice está fuera de límites.
<i>obtener</i>	Devuelve el elemento referenciado por <i>actual</i> , o lanza una excepción de tipo const char * si la lista está vacía. No tiene parámetros.
<i>obtener(i)</i>	Devuelve el elemento de la posición <i>i</i> , o lanza una excepción de tipo const char * si la lista está vacía o el índice está fuera de límites. Tiene un parámetro correspondiente a la posición <i>i</i> del objeto que se desea obtener.
<i>operador []</i> <i>modificar</i>	Igual que <i>obtener(i)</i> . Establece nuevos datos para el elemento <i>actual</i> . Tiene un parámetro que es una referencia al nuevo objeto de tipo <i>T</i> . No devuelve ningún valor.

A continuación se presenta el código correspondiente a la definición de la clase *CListaCircularDE<T>*:

```
// ListaCircularDE.h - Declaración de la plantilla CListaCircularDE
#ifdef _LISTCIRCULARDE_H_
#define _LISTCIRCULARDE_H_

////////////////////////////////////
// Lista circular doblemente enlazada
//
```

```

template<class T>
class CListaCircularDE
{
private:
    // Elemento de una lista circular doblemente enlazada
    class CElemento
    {
        // Atributos
    public:
        T datos;
        CElemento *anterior; // anterior elemento
        CElemento *siguiente; // siguiente elemento
    // Métodos
    public:
        CElemento() { anterior = siguiente = 0; } // constructor
        CElemento(T& d, CElemento *s) // constructor
        {
            datos = d;
            anterior = siguiente = s;
        }
    };
    CElemento *ultimo; // apunta al último elemento
    CElemento *actual; // apunta al último elemento accedido
    long numeroDeElementos; // número de elementos
    long posicion; // posición del elemento actual
public:
    CListaCircularDE();
    ~CListaCircularDE();
    int tamanyo() const;
    void insertar(T& obj);
    T borrar();
    void irAlSiguiente();
    void irAlAnterior();
    void irAlPrincipio();
    void irAlFinal();
    bool irAl(long i);
    T& obtener();
    T& obtener(long i);
    T& operator[](long i);
    void modificar(T& NuevosDatos);
};
////////////////////////////////////
#include "ListaCircularDE.cpp"

#endif // _LISTCIRCULARDE_H_

// ListaCircularDE.cpp -Definición de la plantilla CListaCircularDE
////////////////////////////////////
template<class T>
CListaCircularDE<T>::CListaCircularDE() // constructor
{

```

```
    actual = ultimo = 0;
    numeroDeElementos = 0L;
    posicion = -1L; // la posición del primer elemento será la 0
}
```

```
template<class T>
```

```
CListaCircularDE<T>::~~CListaCircularDE()
```

```
{
    // Borrar los elementos de la lista
    if (numeroDeElementos == 0) return;
    CElemento *q = ultimo->siguiente; // primer elemento
    while (q != ultimo)
    {
        ultimo->siguiente = q->siguiente;
        delete q;
        q = ultimo->siguiente;
    }
    delete ultimo;
}
```

```
template<class T>
```

```
int CListaCircularDE<T>::tamanyo() const
```

```
{
    // Obtener el número de elementos de la lista
    return numeroDeElementos;
}
```

```
template<class T>
```

```
void CListaCircularDE<T>::insertar(T& obj)
```

```
{
    // Añade un nuevo elemento a la lista a continuación
    // del elemento actual; el nuevo elemento pasa a ser el
    // actual.
    CElemento *q;

    if (numeroDeElementos == 0) // lista vacía
    {
        ultimo = new CElemento();

        // Las dos líneas siguientes inician una lista circular.
        ultimo->anterior = ultimo;
        ultimo->siguiente = ultimo;
        ultimo->datos = obj; // asignar datos.
        actual = ultimo;
        posicion = 0L; // ya hay un elemento en la lista.
    }
    else // existe una lista
    {
        q = new CElemento();

        // Insertar el nuevo elemento después del actual.
    }
}
```



```

    actual->siguiente->anterior = q;
    q->siguiente = actual->siguiente;
    actual->siguiente = q;
    q->anterior = actual;
    q->datos = obj;

    // Actualizar parámetros.
    posicion++;

    // Si el elemento actual es el último, el nuevo elemento
    // pasa a ser el actual y el último.
    if( actual == ultimo )
        ultimo = q;

    actual = q; // el nuevo elemento pasa a ser el actual.
} // fin else
numeroDeElementos++; //incrementar en uno el número de elementos.
}

```

```

template<class T>

```

```

T CListaCircularDE<T>::borrar()

```

```

{
    // El método borrar devuelve los datos del elemento
    // apuntado por actual y lo elimina de la lista
    CElemento *q;
    T obj;

    if(numeroDeElementos == 0 ) throw "lista vacía\n";

    if( actual == ultimo ) // se trata del último elemento.
    {
        if( numeroDeElementos == 1L ) // hay sólo un elemento
        {
            obj = ultimo->datos;
            delete ultimo;
            ultimo = actual = 0;
            numeroDeElementos = 0L;
            posicion = -1L;
        }
        else // hay más de un elemento
        {
            actual = ultimo->anterior;
            ultimo->siguiente->anterior = actual;
            actual->siguiente = ultimo->siguiente;
            obj = ultimo->datos;
            delete ultimo;
            ultimo = actual;
            posicion--;
            numeroDeElementos--;
        } // fin del bloque else
    } // fin del bloque if( actual == ultimo )
}

```

```
else // el elemento a borrar no es el último
{
    q = actual->siguiente;
    actual->anterior->siguiente = q;
    q->anterior = actual->anterior;
    obj = actual->datos;
    delete actual;
    actual = q;
    numeroDeElementos--;
}
return obj;
}

template<class T>
void CListaCircularDE<T>::irAlSiguiente()
{
    // Avanza la posición actual al siguiente elemento.
    if (posicion < numeroDeElementos - 1)
    {
        actual = actual->siguiente;
        posicion++;
    }
}

template<class T>
void CListaCircularDE<T>::irAlAnterior()
{
    // Retrasa la posición actual al elemento anterior.
    if (posicion > 0L)
    {
        actual = actual->anterior;
        posicion--;
    }
}

template<class T>
void CListaCircularDE<T>::irAlPrincipio()
{
    // Hace que la posición actual sea el principio de la lista.
    actual = ultimo->siguiente;
    posicion = 0L;
}

template<class T>
void CListaCircularDE<T>::irAlFinal()
{
    // El final de la lista es ahora la posición actual.
    actual = ultimo;
    posicion = numeroDeElementos - 1;
}
```

```

template<class T>
bool CListaCircularDE<T>::irAl(long i)
{
    // Posicionarse en el elemento i
    long numeroDeElementos = tamano();
    if (i >= numeroDeElementos || i < 0) return false;
    irAlPrincipio();
    // Posicionarse en el elemento i
    for (long n = 0; n < i; n++)
        irAlSiguiente();
    return true;
}

template<class T>
T& CListaCircularDE<T>::obtener()
{
    // El método obtener devuelve la referencia a los datos
    // asociados con el elemento actual.
    if (numeroDeElementos == 0 ) throw "lista vacía\n";
    return actual->datos;
}

template<class T>
T& CListaCircularDE<T>::obtener(long i)
{
    // El método obtener devuelve la referencia a los datos
    // asociados con el elemento de índice i.
    if (!irAl(i)) throw "índice fuera de límites\n";
    return obtener();
}

template<class T>
T& CListaCircularDE<T>::operator[](long i)
{
    return obtener(i);
}

template<class T>
void CListaCircularDE<T>::modificar(T& NuevosDatos)
{
    // El método modificar establece nuevos datos para el
    // elemento actual.
    if(numeroDeElementos == 0) return; // lista vacía

    actual->datos = NuevosDatos;
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

Cuando se declara un objeto de la clase *CListaCircularDE*, se ejecuta el constructor de la misma que realiza las siguientes operaciones:

- Crea una lista vacía (*ultimo* = *actual* = 0). En todo momento, el último elemento de la lista está apuntado por *ultimo*, y *actual* apunta al elemento sobre el que se realizará la siguiente operación.
- Asigna un valor 0 al atributo *numeroDeElementos* y un valor -1 a *posicion*; el valor de este atributo pasará a ser 0 cuando se añada el primer elemento.

El método *insertar* de la clase *CListaCircularDE<T>* añade un elemento a la lista a continuación del elemento *actual*. Contempla dos casos: que la lista esté vacía, o que la lista ya exista. El elemento insertado pasa a ser el elemento *actual*, y si se añade al final, éste pasa a ser el *ultimo* y el *actual*. Añadir un elemento implica realizar los enlaces con el anterior y siguiente elementos y actualizar los parámetros *actual*, *numeroDeElementos* y *ultimo*, si procede.

El método *borrar* devuelve una referencia al objeto de datos asociado con el elemento *actual*, elemento que será eliminado cuando finalice la ejecución del método. Contempla dos casos: que el elemento a borrar sea el último o que no lo sea. Si el elemento a borrar es el *ultimo*, y sólo quedaba éste, los atributos de la lista deben iniciarse igual que lo hizo el constructor; si quedaban más de uno, el que es ahora el nuevo *ultimo* pasa a ser también el elemento *actual*. Si el elemento a borrar no era el último, el elemento siguiente al eliminado pasa a ser el elemento *actual*. El método lanza una excepción de tipo **const char *** si la lista está vacía.

Para el resto de los métodos es suficiente con la explicación dada al principio de este apartado, además de en el código.

Ejemplo

El siguiente ejemplo muestra cómo utilizar la clase *CListaCircularDE<T>*. Primeramente creamos un objeto *lcde*, correspondiente a una lista circular doblemente enlazada, en la que cada elemento almacenará una referencia a un objeto *CDatos*; y a continuación realizamos varias operaciones de inserción, movimiento y borrado, para finalmente visualizar los elementos de la lista y comprobar si los resultados son los esperados.

```
// Test.cpp - Crear una lista circular doblemente enlazada
#include <iostream>
#include "ListaCircularDE.h"
#include "datos.h"
#include "leerdatos.h"
using namespace std;

bool mostrarElemento(int i, CListaCircularDE<CDatos>& lista)
{
    // Mostrar un elemento
```

```
try
{
    CDatos obj = lista.obtener(i);
    cout << i << ".- " << obj.obtenerNombre() << ' '
        << obj.obtenerNota() << '\n';
}
catch(const char *s)
{
    cout << s;
    return false;
}
return true;
}

void mostrarLista(CListaCircularDE<CDatos>& lista)
{
    // Mostrar todos los elementos de la lista
    long i = 0, tam = lista.tamanyo();
    while (i < tam)
    {
        mostrarElemento(i, lista); // mostrar el elemento i de lista
        i++;
    }
    if (tam == 0) cout << "lista vacía\n";
}

int main()
{
    // Crear una lista vacía
    CListaCircularDE<CDatos> lcde;

    // Insertar elementos
    CDatos obj;
    obj = CDatos("alumno1", 7.8); lcde.insertar(obj);
    obj = CDatos("alumno2", 6.5); lcde.insertar(obj);
    obj = CDatos("alumno3", 10); lcde.insertar(obj);
    obj = CDatos("alumno4", 8.6); lcde.insertar(obj);

    // Ir al elemento de la posición 2 y borrarlo
    lcde.irAl(2);
    lcde.borrar();

    // Ir al elemento de la posición 1
    // para añadir otro a continuación
    lcde.irAl(1);
    obj = CDatos("nuevo alumno3", 9.5); lcde.insertar(obj);

    // Ir al final e insertar un nuevo elemento
    lcde.irAlFinal();
    obj = CDatos("alumno5", 8.5); lcde.insertar(obj);
}
```

```

// Ir al anterior y modificarlo
lcde.irAlAnterior();
obj = CDatos("alumno4", 5.5); lcde.modificar(obj);

// Mostrar la lista
cout << "\nLista:\n";
mostrarLista(lcde);

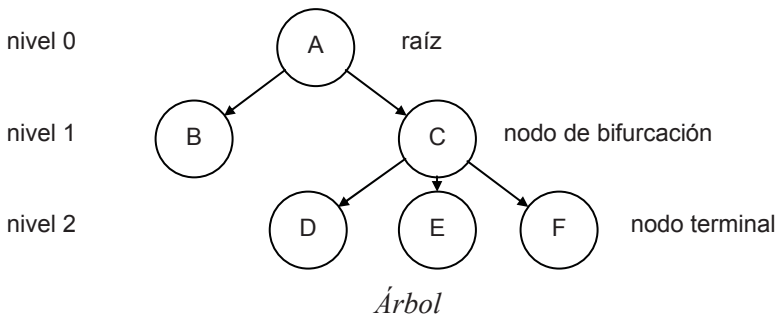
// Mostrar el elemento de la posición i
int i = 0;
do
{
    cout << "posición del elemento (entre 0 y " << lcde.tamano()
        << "): ";
    leerDato(i); // leer la posición
}
while(!mostrarElemento(i, lcde)); // mostrar el elemento i de lcde
}

```

ÁRBOLES

Un árbol es una estructura no lineal formada por un conjunto de nodos y un conjunto de ramas.

En un árbol existe un nodo especial denominado *raíz*. Así mismo, un nodo del que sale alguna rama recibe el nombre de *nodo de bifurcación* o *nodo rama* y un nodo que no tiene ramas recibe el nombre de *nodo terminal* o *nodo hoja*.



De un modo más formal, diremos que un árbol es un conjunto finito de uno o más nodos tales que:

- a) Existe un nodo especial llamado *raíz* del árbol, y
- b) los nodos restantes están agrupados en $n > 0$ conjuntos disjuntos A_1, \dots, A_n , cada uno de los cuales es a su vez un árbol que recibe el nombre de *subárbol de la raíz*.

Evidentemente, la definición dada es recursiva; es decir, hemos definido un árbol como un conjunto de árboles, que es la forma más apropiada de definirlo.

De la definición se desprende que cada nodo de un árbol es la raíz de algún subárbol contenido en la totalidad del mismo.

El número de ramas de un nodo recibe el nombre de *grado* del nodo.

El nivel de un nodo respecto al nodo raíz se define diciendo que la raíz tiene nivel 0 y cualquier otro nodo tiene un nivel igual a la distancia de ese nodo al nodo raíz. El máximo de los niveles se denomina *profundidad* o *altura* del árbol.

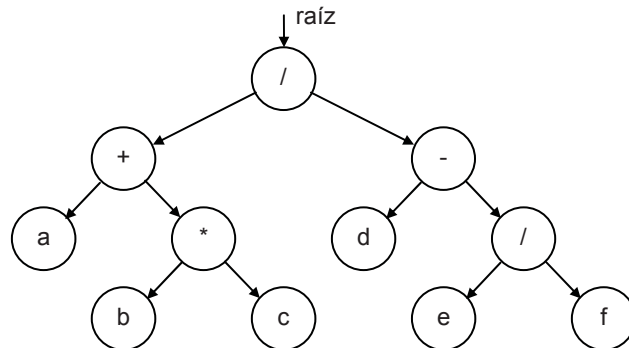
Es útil limitar los árboles en el sentido de que cada nodo sea a lo sumo de grado 2. De esta forma cabe distinguir entre subárbol izquierdo y subárbol derecho de un nodo. Los árboles así formados se denominan *árboles binarios*.

Árboles binarios

Un árbol binario es un conjunto finito de nodos que consta de un *nodo raíz* que tiene dos subárboles binarios denominados *subárbol izquierdo* y *subárbol derecho*.

Las expresiones algebraicas, debido a que los operadores que intervienen son operadores binarios, nos dan un ejemplo de estructura en árbol binario. La figura siguiente nos muestra un árbol que corresponde a la expresión aritmética:

$$(a + b * c) / (d - e / f)$$



Expresión algebraica

El árbol binario es una estructura de datos muy útil cuando el tamaño de la estructura no se conoce, se necesita acceder a sus elementos ordenadamente, la ve-

localidad de búsqueda es importante o el orden en el que se insertan los elementos es casi aleatorio.

En definitiva, un árbol binario es una colección de objetos (nodos del árbol) cada uno de los cuales contiene datos o un puntero a los datos, un puntero a su subárbol izquierdo y un puntero a su subárbol derecho. Según lo expuesto, la estructura de datos representativa de un nodo puede ser de la forma siguiente:

```
// Nodo de un árbol binario
class CNodo
{
    // Atributos
public:
    T datos;           // define el área de datos
    CNodo *izquierdo; // raíz del subárbol izquierdo
    CNodo *derecho;   // raíz del subárbol derecho
    // Métodos
public:
    CNodo() { izquierdo = derecho = 0; } // constructor
};
```

La definición dada de árbol binario sugiere una forma natural de representar árboles binarios en un ordenador. Una variable *raíz* referenciará el árbol y cada nodo del árbol será un objeto de la clase *CNodo*. Esto es, la declaración genérica de un árbol binario puede ser así:

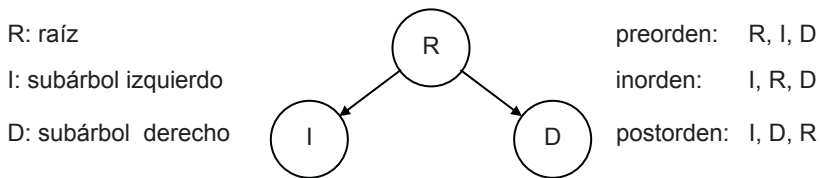
```
class CArbolBinario
{
private:
    // Nodo de un árbol binario
    class CNodo
    {
        // Atributos
public:
        T datos;           // define el área de datos
        CNodo *izquierdo; // raíz del subárbol izquierdo
        CNodo *derecho;   // raíz del subárbol derecho
        // Métodos
public:
        CNodo() { izquierdo = derecho = 0; } // constructor
    };
    // Atributos del árbol binario
    CNodo *raiz; // raíz del árbol
public:
    // Métodos del árbol binario
    CArbolBinario() { raiz = 0; } // constructor
    // ...
};
```


Si el árbol está vacío, *raiz* es igual a cero; en otro caso, *raiz* es un puntero al nodo raíz del árbol y, según se puede observar en el código anterior, este nodo tiene tres atributos: una variable que define los datos, un puntero a su subárbol izquierdo y otro a su subárbol derecho.

Formas de recorrer un árbol binario

Observe la figura “expresión algebraica” mostrada anteriormente. Partiendo del nodo raíz, ¿qué orden seguimos para poder evaluar la expresión que representa el árbol? Hay varios algoritmos para el manejo de estructuras en árbol y un proceso que generalmente se repite en estos algoritmos es el de recorrido de un árbol. Este proceso consiste en examinar sistemáticamente los nodos de un árbol, de forma que cada nodo sea visitado solamente una vez.

Básicamente se pueden utilizar tres formas para recorrer un árbol binario: *preorden*, *inorden* y *postorden*. Cuando se utiliza la forma *preorden*, primero se visita la raíz, después el subárbol izquierdo y por último el subárbol derecho; en cambio, si se utiliza la forma *inorden*, primero se visita el subárbol izquierdo, después la raíz y por último el subárbol derecho; y si se utiliza la forma *postorden*, primero se visita el subárbol izquierdo, después el subárbol derecho y por último la raíz.



Formas de recorrer un árbol

Evidentemente, las definiciones dadas son definiciones recursivas, ya que recorrer un árbol utilizando cualquiera de ellas implica recorrer sus subárboles empleando la misma definición.

Si se aplican estas definiciones al árbol binario de la figura “expresión algebraica” mostrada anteriormente, se obtiene la siguiente solución:

Preorden:	/ + a * b c - d / e f
Inorden:	a + b * c / d - e / f
Postorden:	a b c * + d e f / - /

El recorrido en preorden produce la notación *prefija*; el recorrido en inorden produce la notación *convencional* y el recorrido en postorden produce la notación *postfija* o *inversa*.

Los nombres de preorden, inorden y postorden derivan del lugar en el que se visita la raíz con respecto a sus subárboles. Estas tres formas se exponen a continuación como tres métodos recursivos de la clase *CArbolBinario*, con un único parámetro *r* que representa la raíz del árbol cuyos nodos se quieren visitar.

```
// CArbolBinario.h - Declaración de la clase CArbolBinario
#ifndef !defined( _CARBOLBINARIO_H_ )
#define _CARBOLBINARIO_H_

template<class T>
class CArbolBinario
{
private:
    // Nodo de un árbol binario
    class CNodo
    {
    // Atributos
    public:
        T datos;           // define el área de datos
        CNodo *izquierdo; // raíz del subárbol izquierdo
        CNodo *derecho;   // raíz del subárbol derecho
    // Métodos
    public:
        CNodo() { izquierdo = derecho = 0; } // constructor
    };
    // Atributos del árbol binario
    CNodo *raiz; // raíz del árbol
public:
    // Métodos del árbol binario
    CArbolBinario() { raiz = 0; } // constructor
    void preorden(CNodo r);
    void inorden(CNodo r);
    void postorden(CNodo r);
};
////////////////////////////////////
#include "CArbolBinario.cpp"
#endif // _CARBOLBINARIO_H_
```

```
// CArbolBinario.cpp - Definición de la clase CArbolBinario
////////////////////////////////////
template<class T>
void CArbolBinario<T>::preorden(CNodo r)
{
    if ( r != 0 )
    {
        // Escribir aquí las operaciones a realizar
        // con el nodo apuntado r
        preorden( r->izquierdo); // se visita el subárbol izquierdo
    }
}
```

```

        preorden( r->derecho); // se visita el subárbol derecho
    }
}

template<class T>
void CArbolBinario<T>::inorden(CNodo r)
{
    if ( r != 0 )
    {
        inorden( r->izquierdo); // se visita el subárbol izquierdo
        // Escribir aquí las operaciones a realizar
        // con el nodo apuntado r
        inorden( r->derecho); // se visita el subárbol derecho
    }
}

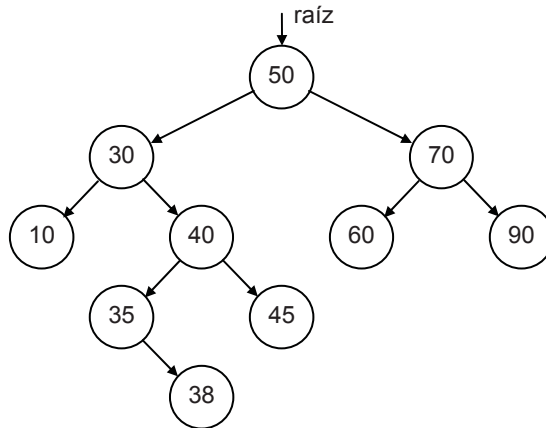
template<class T>
void CArbolBinario<T>::postorden(CNodo r)
{
    if ( r != 0 )
    {
        postorden( r->izquierdo); // se visita el subárbol izquierdo
        postorden( r->derecho); // se visita el subárbol derecho
        // Escribir aquí las operaciones a realizar
        // con el nodo apuntado r
    }
}
////////////////////////////////////

```

ÁRBOLES BINARIOS DE BÚSQUEDA

Un *árbol binario de búsqueda* es un árbol ordenado; esto es, las ramas de cada nodo están ordenadas de acuerdo con las siguientes reglas: para todo nodo a_i , todas las claves del subárbol izquierdo de a_i son menores que la clave de a_i y todas las claves del subárbol derecho de a_i son mayores que la clave de a_i .

Con un árbol de estas características, encontrar si un nodo de una clave determinada existe o no es una operación muy sencilla. Por ejemplo, observando la figura siguiente, localizar la clave 35 es aplicar la definición de árbol de búsqueda; esto es, si la clave buscada es menor que la clave del nodo en el que estamos, pasamos al subárbol izquierdo de este nodo para continuar la búsqueda, y si es mayor, pasamos al subárbol derecho. Este proceso continúa hasta encontrar la clave o hasta llegar a un subárbol vacío, árbol cuya raíz tiene un valor cero.



Árbol binario de búsqueda

En C++ podemos automatizar el proceso de implementar un árbol binario de búsqueda diseñando una clase *CarbolBinB* (Clase *Arbol Binario de Búsqueda*) que proporcione los atributos y métodos necesarios para crear cada nodo del árbol, así como para permitir el acceso a los mismos. La clase que diseñamos a continuación cubre estos objetivos.

Clase **CarbolBinB<T>**

La clase *CarbolBinB<T>* que vamos a implementar incluirá un atributo protegido *raiz* para referenciar la raíz del árbol y tres constantes públicas relacionadas con las posibles situaciones que se pueden dar al operar con un nodo: *CORRECTO*, *YA_EXISTE* y *NO_EXISTE*. El atributo *raiz* valdrá cero cuando el árbol esté vacío. Así mismo incluye una clase interna, *CNodo*, que define la estructura de los nodos, y los métodos indicados en la tabla siguiente:

Método	Significado
<i>CarbolBinB</i>	Es el constructor. Crea un árbol vacío (<i>raiz</i> = 0).
<i>~CarbolBinB</i>	Es el destructor. Invoca al método <i>borrarArbol</i> para borrar todo el árbol.
<i>borrarArbol</i>	Método recursivo que permite liberar la memoria asignada a cada nodo del árbol, recorriéndolo según la forma <i>postorden</i> . No libera la memoria asignada al área de datos de cada nodo (si es que se asignó), tarea que corresponde a la aplicación que utilice esta interfaz.

Método	Significado
<i>buscar</i>	Busca un nodo determinado en el árbol. Tiene un parámetro para almacenar una referencia de tipo <i>T</i> a los datos que permitirán localizar el nodo en el árbol. Devuelve un puntero al área de datos del nodo o bien cero si el árbol está vacío o no existe un nodo con esos datos.
<i>insertar</i>	Inserta un nodo en el árbol binario basándose en la definición de árbol binario de búsqueda. Tiene un parámetro que es una referencia al objeto de tipo <i>T</i> a añadir. Devuelve la constante <i>CORRECTO</i> si la operación de inserción se realiza satisfactoriamente o <i>YA_EXISTE</i> , si el nodo con esos datos ya existe.
<i>borrar</i>	Borra un nodo de un árbol binario de búsqueda. Tiene un parámetro para almacenar una referencia a los datos de tipo <i>T</i> que permitirán localizar en el árbol el nodo que se desea borrar. Devuelve la constante <i>CORRECTO</i> si la operación de borrado se realiza satisfactoriamente o <i>NO_EXISTE</i> , si el nodo con esos datos no existe.
<i>inorden</i>	Recorre el árbol binario para el que es invocado utilizando la forma <i>inorden</i> . Tiene dos parámetros: el primero especifica la dirección del nodo a partir del cual se realizará la visita; este parámetro sólo será tenido en cuenta si el segundo es false , porque si es true se asume que el primero toma como valor la raíz del árbol. No devuelve ningún valor.
<i>comparar</i>	Método virtual que debe ser redefinido por el usuario en una clase derivada para especificar el tipo de comparación que se desea realizar con dos nodos del árbol. Debe de devolver un entero indicando el resultado de la comparación (-1, 0 ó 1 si <i>nodo1</i> < <i>nodo2</i> , <i>nodo1</i> == <i>nodo2</i> , o <i>nodo1</i> > <i>nodo2</i> , respectivamente). Este método es invocado por los métodos <i>insertar</i> , <i>borrar</i> y <i>buscar</i> .
<i>proceso</i>	Método virtual que debe ser redefinido por el usuario en una clase derivada para especificar las operaciones que se desean realizar con el nodo visitado. Es invocado por el método <i>inorden</i> .

A continuación se presenta el código correspondiente a la definición de la clase *CarbolBinB<T>*:

```
// CarbolBinB.h - Declaración de la clase CarbolBinB<T>
#ifdef !defined( _CARBOLBINB_H_ )
#define _CARBOLBINB_H_

////////////////////////////////////
```

```
// Clase abstracta: árbol binario de búsqueda. Para utilizar
// los métodos proporcionados por esta clase, tendremos que
// crear una clase derivada de ella y redefinir los métodos:
// comparar y procesar.
//
// Posibles situaciones que se pueden dar respecto de un nodo
enum cods_error
{
    NO_EXISTE = 0, YA_EXISTE = 1, CORRECTO = 2
};

template<class T>
class CArbolBinB
{
    // Atributos del árbol binario
protected:
    // Nodo de un árbol binario
    class CNodo
    {
        // Atributos
    public:
        T datos;           // define el área de datos
        CNodo *izquierdo; // raíz del subárbol izquierdo
        CNodo *derecho;   // raíz del subárbol derecho
        // Métodos
    public:
        CNodo() { izquierdo = derecho = 0; } // constructor
    };
    CNodo *raiz; // raíz del árbol

// Métodos del árbol binario
public:
    CArbolBinB() { raiz = 0; } // constructor
    virtual ~CArbolBinB();     // destructor
    void borrarArbol(CNodo *r); // borra todo el árbol

    // El método siguiente debe ser redefinido en una clase
    // derivada para que permita comparar dos nodos del árbol
    // por el atributo que necesitamos en cada momento.
    // Devuelve un valor menor, igual o mayor que cero, dependiendo
    // de que obj1 sea menor, igual o mayor que obj2.
    virtual int comparar(T& obj1, T& obj2) = 0;

    // El método siguiente debe ser redefinido en una clase
    // derivada para que permita especificar las operaciones
    // que se deseen realizar con el nodo visitado.
    virtual void procesar(T& obj) = 0;

    T *buscar(T& obj);
    int insertar(T& obj);
    int borrar(T& obj);
};
```

```

        void inorden(CNodo *r = 0, bool nodoRaiz = true);
    };
    //////////////////////////////////////
#include "CArbolBinB.cpp"
#endif // _CARBOLBINB_H_

// CArbolBinB.cpp - Definición de la clase CArbolBinB<T>
//
template<class T>
CArbolBinB<T>::~CArbolBinB() // destructor
{
    borrarArbol(raiz);
    raiz = 0;
}

template<class T>
void CArbolBinB<T>::borrarArbol(CNodo *r)
{
    if ( r != 0 )
    {
        borrarArbol(r->izquierdo); // se visita el subárbol izquierdo
        borrarArbol(r->derecho);   // se visita el subárbol derecho
        delete r;
    }
}

template<class T>
T *CArbolBinB<T>::buscar(T& obj)
{
    // ...
}

template<class T>
int CArbolBinB<T>::insertar(T& obj)
{
    // ...
}

template<class T>
int CArbolBinB<T>::borrar(T& obj)
{
    // ...
}

template<class T>
void CArbolBinB<T>::inorden(CNodo *r, bool nodoRaiz)
{
    // El método recursivo inorden visita los nodos del árbol
    // utilizando la forma inorden; esto es, primero se visita
    // el subárbol izquierdo, después se visita la raíz, y por
    // último, el subárbol derecho.
}

```

```

// Si el segundo parámetro es true, la visita comienza
// en la raíz independientemente del primer parámetro.

CNode *actual = 0;
if (nodoRaiz)
    actual = raiz; // partir de la raíz
else
    actual = r; // partir de un nodo cualquiera
if (actual != 0)
{
    inorden(actual->izquierdo, false); // visitar subárbol izq.
    // Procesar los datos del nodo visitado
    procesar(actual->datos);
    inorden(actual->derecho, false); // visitar subárbol dcho.
}
}
}
/////////////////////////////////////////////////////////////////

```

Buscar un nodo en el árbol

El método *buscar* cuyo código se muestra a continuación permite acceder a los datos de un nodo del árbol.

```

template<class T>
T *CArbolBinB<T>::buscar(T& obj)
{
    // El método buscar permite acceder a un determinado nodo.
    CNode *actual = raiz;
    int nComp = 0;

    // Buscar un nodo que tenga asociados los datos dados por obj
    while (actual != 0)
    {
        if (( nComp = comparar( obj, actual->datos ) ) == 0)
            return &actual->datos; // CORRECTO (nodo encontrado)
        else if (nComp < 0) // buscar en el subárbol izquierdo
            actual = actual->izquierdo;
        else // buscar en el subárbol derecho
            actual = actual->derecho;
    }
    return 0; // NO_EXISTE
}

```

El parámetro *obj* se refiere al objeto de datos, que suponemos definido por un nodo del árbol, al que deseamos acceder. Este método devuelve un puntero nulo si el objeto referenciado por *obj* no se localiza en el árbol, o bien un puntero al objeto de datos del nodo localizado.

Por definición de árbol de búsqueda, sabemos que sus nodos tienen que estar ordenados utilizando como clave alguno de los atributos de *obj*. Según esto, el método *buscar* se escribe aplicando estrictamente esa definición; esto es, si la clave buscada es menor que la clave del nodo en el que estamos, continuamos la búsqueda en su subárbol izquierdo y si es mayor, entonces continuamos la búsqueda en su subárbol derecho. Este proceso continúa hasta encontrar la clave o bien hasta llegar a un subárbol vacío (subárbol cuya raíz tiene un valor cero).

Para saber si una clave es igual, menor o mayor que otra invocaremos al método *comparar* pasando como argumentos los objetos de datos que contienen los atributos que se desean comparar. Como tales atributos, dependiendo de la aplicación, pueden ser bien de algún tipo numérico o bien de tipo alfanumérico o alfabético, la implementación de este método hay que posponerla al diseño de la aplicación que utilice esta clase, razón por la que *comparar* ha sido definido como un método virtual puro. Para ello, como veremos un poco más adelante, derivaremos una nueva clase de ésta, y redefiniremos este método.

Insertar un nodo en el árbol

El método *insertar* cuyo código se muestra a continuación permite añadir un nodo que aún no existe en el árbol.

```
template<class T>

int CArbolBinB<T>::insertar(T& obj)
{
    // El método insertar permite añadir un nodo que aún no está
    // en el árbol.
    CNodo *ultimo = 0, *actual = raiz;
    int nComp = 0;

    // Comienza la búsqueda para verificar si ya hay un nodo con
    // estos datos en el árbol
    while (actual != 0)
    {
        if ((nComp = comparar( obj, actual->datos )) == 0)
            break; // se encontró el nodo
        else
        {
            ultimo = actual;
            if (nComp < 0) // buscar en el subárbol izquierdo
                actual = actual->izquierdo;
            else // buscar en el subárbol derecho
                actual = actual->derecho;
        }
    }
}
```

```

if (actual == 0) // no se encontró el nodo, añadirlo
{
    CNodo *nuevoNodo = new CNodo();
    nuevoNodo->datos = obj;
    nuevoNodo->izquierdo = nuevoNodo->derecho = 0;

    // El nodo a añadir pasará a ser la raíz del árbol total si
    // éste está vacío, del subárbol izquierdo de "ultimo" si la
    // comparación fue menor o del subárbol derecho de "ultimo" si
    // la comparación fue mayor.
    if (ultimo == 0) // árbol vacío
        raiz = nuevoNodo;
    else if ( nComp < 0 )
        ultimo->izquierdo = nuevoNodo;
    else
        ultimo->derecho = nuevoNodo;

    return CORRECTO;
} // fin del bloque if ( actual == 0 )
else // el nodo ya existe en el árbol
    return YA_EXISTE;
}

```

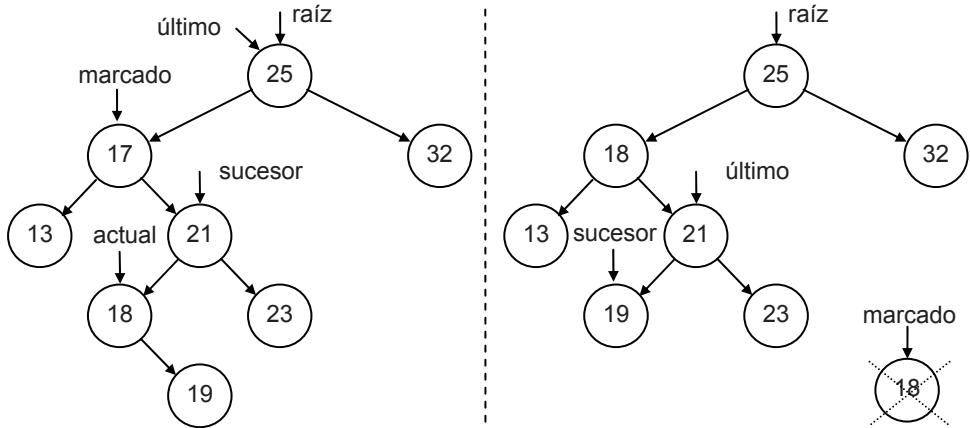
El parámetro *obj* se refiere al objeto de datos al que hará referencia el nodo que se añadirá al árbol. Devuelve un valor *CORRECTO* si la operación de insertar se ejecuta con éxito, y *YA_EXISTE* si ya hay un nodo con los datos referenciados por *obj*.

Este método lo primero que hace es verificar si ya hay un nodo con estos datos en el árbol (para realizar esta operación se sigue el mismo proceso descrito en el método *buscar*), en cuyo caso, como ya se indicó en el párrafo anterior, lo notificará. Si ese nodo no se encuentra, el proceso de búsqueda nos habrá conducido hasta un nodo terminal, posición donde lógicamente debe añadirse el nuevo nodo que almacenará la referencia *obj* a los datos.

Borrar un nodo del árbol

A continuación se estudia el problema de borrar un determinado nodo de un árbol que tiene las claves ordenadas. Este proceso es una tarea fácil si el nodo a borrar es un nodo terminal o si tiene un único descendiente. La dificultad se presenta cuando deseamos borrar un nodo que tiene dos descendientes (en la figura, 17), ya que con un solo puntero no se puede apuntar en dos direcciones. En este caso, el nodo a borrar será reemplazado, bien por el nodo más a la derecha (13) de su subárbol izquierdo (nodo raíz, 13) o bien por el nodo más a la izquierda (18) de su subárbol derecho (nodo raíz, 21). Obsérvese que la forma en la que se ha elegido

el nodo empleado en la sustitución, que después se eliminará, conserva la definición de árbol de búsqueda en el árbol resultante.



Borrar el nodo con clave 17

En la figura anterior, la variable *actual* apunta a la raíz del subárbol en el que continúa la búsqueda; inicialmente su valor es *raíz*. La variable *marcado* apunta al nodo a borrar una vez localizado. La variable *ultimo* apunta finalmente al último nodo visitado antes del nodo a borrar.

Para encontrar el nodo a borrar (17), se desciende por el árbol aplicando los criterios que lo definen. Una vez localizado, comprobamos si se corresponde con:

1. Un nodo terminal (no tiene descendientes).
2. Un nodo que no tiene subárbol izquierdo.
3. Un nodo que no tiene subárbol derecho.
4. Un nodo que tiene subárbol izquierdo y derecho.

En los casos 1, 2 y 3 se elimina el nodo apuntado por *marcado* y se actualizan los enlaces del nodo apuntado por *ultimo* para establecer sus descendientes

En el caso 4, decidimos aplicar el algoritmo de sustitución a partir del subárbol derecho del nodo a borrar. Esto significa descender por este subárbol buscando el nodo más a la izquierda (18), que quedará apuntado por *actual*. Después se copian los datos de *actual* en *marcado*, con la intención de que *actual* pase a ser el nodo a borrar y convertir, de esta forma, el caso 4 en un caso 2 que es más sencillo de tratar; por lo tanto, si *actual* no tiene subárbol derecho, el puntero *marcado*->*derecho* debe pasar a valer 0, *marcado* debe apuntar al nuevo nodo a borrar; *ultimo*, al nodo anterior y *sucesor*, al descendiente de *marcado*. A partir de

aquí, la ejecución del método continúa como si se tratara de un caso 2. Finalmente, el nodo apuntado por *marcado* es eliminado.

```
template<class T>
int CArbolBinB<T>::borrar(T& obj)
{
    // El método borrar permite eliminar un nodo del árbol.
    CNode *ultimo = 0, *actual = raiz;
    CNode *marcado = 0, *sucesor = 0;
    int nAnteriorComp = 0, nComp = -1;
    // nComp = -1 --> nodo más a la izquierda del subárbol derecho

    // Comienza la búsqueda para verificar si hay un nodo con
    // estos datos en el árbol.
    while (actual != 0)
    {
        nAnteriorComp = nComp; // resultado de la comparación anterior
        if (( nComp = comparar( obj, actual->datos )) == 0)
            break; // se encontró el nodo
        else
        {
            ultimo = actual;
            if ( nComp < 0 ) // buscar en el subárbol izquierdo
                actual = actual->izquierdo;
            else // buscar en el subárbol derecho
                actual = actual->derecho;
        }
    }
    // fin del bloque while ( actual != 0 )

    if (actual != 0) // se encontró el nodo
    {
        marcado = actual;
        if (( actual->izquierdo == 0 && actual->derecho == 0 ))
            // se trata de un nodo terminal (no tiene descendientes)
            sucesor = 0;
        else if (actual->izquierdo == 0) // nodo sin subárbol izquierdo
            sucesor = actual->derecho;
        else if (actual->derecho == 0) // nodo sin subárbol derecho
            sucesor = actual->izquierdo;
        else // nodo con subárbol izquierdo y derecho
        {
            // Puntero al subárbol derecho del nodo a borrar
            sucesor = actual = actual->derecho;
            // Descender al nodo más a la izquierda en el subárbol
            // derecho de este nodo (el de valor más pequeño)
            while (actual->izquierdo != 0)
                actual = actual->izquierdo;
            // Sustituir el nodo a borrar por el nodo más a la izquierda
            // en el subárbol derecho que pasará a ser el nodo a borrar
            marcado->datos = actual->datos;
        }
    }
}
```

```

    if (actual->derecho == 0) marcado->derecho = 0;
    // Identificar el nuevo nodo a borrar
    marcado = actual; // éste es ahora el nodo a borrar
    ultimo = sucesor;
    sucesor = actual->derecho;
}

// Actualizar los enlaces prescindiendo de marcado
if (ultimo != 0)
{
    if (nAnteriorComp < 0)
        ultimo->izquierdo = sucesor;
    else if (nAnteriorComp > 0)
        ultimo->derecho = sucesor;
}
else
    raiz = sucesor;

// Eliminar el nodo apuntado por marcado
delete marcado;
return CORRECTO;
}
else // el nodo buscado no está en el árbol
    return NO_EXISTE;
}

```

Utilización de la clase **CArbolBinB<T>**

La clase *CArbolBinB<T>* es abstracta; por lo tanto, para hacer uso del soporte que proporciona para la construcción y manipulación de árboles binarios de búsqueda, tendremos que derivar una clase de ella y redefinir los métodos virtuales puros heredados: *comparar* y *procesar*. La redefinición de estos métodos está condicionada a la clase de objetos que definen los datos que soporta el árbol.

Como ejemplo, vamos a construir un árbol binario de búsqueda en el que cada nodo se corresponda con un objeto de la clase *CDatos* ya utilizada anteriormente en este mismo capítulo. Esto implica pensar en la clave de ordenación que se utilizará para construir el árbol. En nuestro ejemplo vamos a ordenar los nodos del árbol por el atributo *nombre* de *CDatos*. Se trata entonces de una ordenación alfabética; por tanto, el método *comparar* debe ser redefinido para que devuelva *-1*, *0* ó *1* según sea el *nombre* de un objeto *CDatos* menor, igual o mayor, respectivamente, que el *nombre* del otro objeto con el que se compara.

Pensemos ahora en el proceso que deseamos realizar con cada nodo accedido. En el ejemplo, simplemente nos limitaremos a mostrar los datos *nombre* y *nota*.

Según esto, el método *procesar* obtendrá los datos *nombre* y *nota* del objeto *CDatos* pasado como argumento y los mostrará.

```
// CarbolBinarioDeBusqueda.h - Declaración de la clase
//                                     CarbolBinarioDeBusqueda
#if !defined( _CARBOLBINARIODEBUSQUEDA_H_ )
#define _CARBOLBINARIODEBUSQUEDA_H_
#include "CarbolBinB.h"
#include "datos.h"

////////////////////////////////////
// Clase derivada de la clase abstracta CarbolBinB<T>. Redefine los
// métodos: comparar y procesar.
//
class CarbolBinarioDeBusqueda : public CarbolBinB<CDatos>
{
public:
    int comparar(CDatos& obj1, CDatos& obj2);
    void procesar(CDatos& obj);
};
////////////////////////////////////

#endif // _CARBOLBINARIODEBUSQUEDA_H_

// CarbolBinarioDeBusqueda.cpp - Definición de la clase
//                                     CarbolBinarioDeBusqueda
#include <iostream>
#include <string>
#include "CarbolBinarioDeBusqueda.h"
using namespace std;

////////////////////////////////////
int CarbolBinarioDeBusqueda::comparar(CDatos& obj1, CDatos& obj2)
{
    // Permite comparar dos nodos del árbol por el atributo
    // nombre.
    string str1(obj1.obtenerNombre());
    string str2(obj2.obtenerNombre());
    return str1.compare(str2);
}

void CarbolBinarioDeBusqueda::procesar(CDatos& obj)
{
    // Permite mostrar los datos del nodo visitado.
    string nombre(obj.obtenerNombre());
    double nota = obj.obtenerNota();
    cout << nombre << " " << nota << endl;
}
////////////////////////////////////
```

Ahora puede comprobar de una forma clara que los métodos *comparar* y *procesar* dependen del tipo de objetos que almacenemos en el árbol que construyamos. Por esta razón no pudieron ser implementados en la clase *CArbolBinB<T>*, sino que hay que implementarlos para cada caso particular.

Cuando se declare un objeto de la clase *CArbolBinarioDeBusqueda*, el constructor de esta clase invoca al de su clase base *CArbolBinB<T>*, que creará un árbol vacío (*raiz = 0*). El atributo *raiz* apunta siempre a la raíz del árbol.

Finalmente, escribiremos una aplicación *Test* que, utilizando la clase *CArbolBinarioDeBusqueda*, cree un objeto *arbolbb* correspondiente a un árbol binario de búsqueda en el que cada nodo almacene un objeto *CDatos* que encapsule el nombre de un alumno y la nota de una determinada asignatura que está cursando. Con el fin de probar que todos los métodos proporcionados por la clase funcionan adecuadamente (piense en los métodos heredados y en los redefinidos), la aplicación realizará las operaciones siguientes:

1. Creará un objeto *arbolbb* de la clase *CArbolBinarioDeBusqueda*.
2. Solicitará parejas de datos *nombre* y *nota*, a partir de las cuales construirá los objetos *CDatos* que añadiremos como nodos en el *arbolbb*.
3. Durante la construcción del árbol, permitirá modificar la nota de un alumno ya existente o bien eliminarlo. Para discriminar una operación de otra tomaremos como referencia la nueva nota: si es positiva, entenderemos que deseamos modificar la nota del alumno especificado y si es negativa, que hay que eliminarlo.
4. Finalmente, mostrará los datos almacenados en el árbol para comprobar que todo ha sucedido como esperábamos.

```
// test.cpp - Crear un árbol binario de búsqueda
#include <iostream>
#include <string>
#include "CArbolBinarioDeBusqueda.h"
#include "datos.h"
#include "leerdatos.h"
using namespace std;

int main()
{
    CArbolBinarioDeBusqueda arbolbb;

    // Leer datos y añadirlos al árbol
    string nombre;
    double nota;
    int cod;
```

```

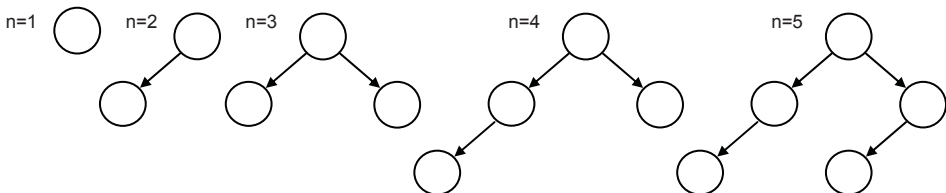
CDatos obj;
cout << "Introducir datos. Finalizar con Ctrl+Z.\n";
cout << "nombre: ";
while (leerDato(nombre))
{
    cout << "nota:  ";
    leerDato(nota);
    obj = CDatos(nombre, nota);
    cod = arbolbb.insertar(obj);
    if (cod == YA_EXISTE)
    {
        // Si ya existe, distinguimos dos casos:
        // 1. nota nueva >= 0; cambiamos la nota
        // 2. nota nueva < 0; borramos el nodo
        if (nota >= 0)
        {
            CDatos *datos = arbolbb.buscar(obj);
            datos->asignarNota(nota);
        }
        else
        {
            arbolbb.borrar(obj);
            cout << "nodo borrado\n";
        }
    }
    cout << "nombre: ";
}
cout << "\n\n";

// Mostrar los nodos del árbol
cout << "\nArbol:\n";
arbolbb.inorden();
}

```

ÁRBOLES BINARIOS PERFECTAMENTE EQUILIBRADOS

Un árbol binario está perfectamente equilibrado si, para todo nodo, el número de nodos en el subárbol izquierdo y el número de nodos en el subárbol derecho difieren como mucho en una unidad.



Árboles perfectamente equilibrados

Como ejemplo, considere el problema de construir un árbol perfectamente equilibrado siendo los valores almacenados en los nodos n objetos de la clase *CDatos* implementada anteriormente en este mismo capítulo. Recuerde que cada objeto de esta clase encapsula el nombre de un alumno y la nota de una determinada asignatura que está cursando.

Esto puede realizarse fácilmente distribuyendo los nodos, según se leen, equitativamente a la izquierda y a la derecha de cada nodo. El proceso recursivo que se indica a continuación es la mejor forma de realizar esta distribución. Para un número dado n de nodos y siendo ni (nodos a la izquierda) y nd (nodos a la derecha) dos enteros, el proceso es el siguiente:

1. Utilizar un nodo para la raíz.
2. Generar el subárbol izquierdo con $ni = n/2$ nodos utilizando la misma regla.
3. Generar el subárbol derecho con $nd = n-ni-1$ nodos utilizando la misma regla.

Cada nodo del árbol consta de los siguientes miembros: *datos*, puntero al subárbol *izquierdo* y puntero al subárbol *derecho*.

```
class CNodo
{
    // Atributos
public:
    int numeroDeNodos; // nodos del subárbol que tiene esta raíz
    T datos;           // define el área de datos
    CNodo *izquierdo; // raíz del subárbol izquierdo
    CNodo *derecho;   // raíz del subárbol derecho
    // Métodos
public:
    CNodo() { izquierdo = derecho = 0; } // constructor
};
```

En C++ podemos automatizar el proceso de implementar un árbol binario perfectamente equilibrado diseñando una clase *CArbolBinE<T>* (Clase *Árbol Binario Equilibrado*) que proporcione los atributos y métodos necesarios para crear cada nodo del árbol, así como para permitir el acceso a los mismos.

Clase *CArbolBinE<T>*

La clase *CArbolBinE<T>* que vamos a implementar incluirá un atributo protegido *raiz* para apuntar a la raíz del árbol. El atributo *raiz* valdrá cero cuando el árbol esté vacío. Así mismo, incluirá la clase interna *CNodo* que define la estructura de los nodos, y los métodos indicados en la tabla siguiente:

Método	Significado
<i>CArbolBinE</i>	Es el constructor. Crea un árbol vacío (<i>raiz = 0</i>).
<i>~CArbolBinE</i>	Es el destructor. Invoca al método <i>borrarArbol</i> para borrar todo el árbol.
<i>borrarArbol</i>	Método recursivo que permite liberar la memoria asignada a cada nodo del árbol, recorriéndolo según la forma <i>postorden</i> . No libera la memoria asignada al área de datos de cada nodo (si es que se asignó), tarea que corresponde a la aplicación que utilice esta interfaz.
<i>construirArbol</i>	Es un método privado que permite construir un árbol binario perfectamente equilibrado. Tiene un parámetro de tipo int que se corresponde con el número de nodos que va a tener el árbol. Devuelve un puntero a la raíz del árbol.
<i>construirArbolEquilibrado</i>	Invoca al método <i>construirArbol</i> pasando como argumento el número de nodos y almacena el valor devuelto por él, en el atributo <i>raiz</i> de la clase (si al usuario se le permitiera invocar a <i>construirArbol</i> tendría que tener acceso al atributo <i>raiz</i> devuelto por este método, de aquí esta solución). No devuelve nada.
<i>obtenerNodo</i>	Busca un nodo determinado en el árbol por posición. Tiene dos parámetros: el índice 0, 1, 2..., según el orden de acceso seguido por la forma <i>inorden</i> (consideramos que la primera posición es la 0), del nodo al que se desea acceder y la raíz del árbol. Devuelve un puntero al área de datos del nodo o 0 si el árbol está vacío o el índice está fuera de los límites.
<i>buscar</i>	Busca un nodo determinado en el árbol. El primer parámetro hace referencia a los datos del nodo a localizar en el árbol y el segundo, opcional, es un puntero a un entero que especifica, inicialmente, la posición del nodo donde se iniciará la búsqueda y, finalmente, la posición del nodo encontrado (consideramos que la primera posición es la 0 referida al orden de acceso según la forma <i>inorden</i>). Devuelve un puntero al área de datos del nodo o bien cero si el árbol está vacío o no existe un nodo con esos datos.
<i>inorden</i>	Recorre un árbol binario utilizando la forma <i>inorden</i> . Tiene dos parámetros: el primero especifica el puntero al nodo a partir del cual se realizará la visita; el valor del primer parámetro sólo será tenido en cuenta si el segundo es false , porque si es true se asume que el primer parámetro es la raíz del árbol. No devuelve ningún valor.

Método	Significado
<i>leerDatos</i>	Método virtual puro que debe ser redefinido por el usuario en una clase derivada para que permita leer los datos a los que hace referencia un nodo del árbol. Devuelve el objeto de datos. Es invocado por el método <i>construirArbol</i> .
<i>comparar</i>	Método virtual puro que debe ser redefinido por el usuario en una clase derivada para especificar el tipo de comparación que se desea realizar con dos nodos del árbol. Debe devolver un entero indicando el resultado de la comparación (-1, 0 ó 1 si <i>nodo1</i> < <i>nodo2</i> , <i>nodo1</i> == <i>nodo2</i> o <i>nodo1</i> > <i>nodo2</i> , respectivamente). Este método es invocado por los métodos <i>insertar</i> , <i>borrar</i> y <i>buscar</i> .
<i>proceso</i>	Método virtual puro que debe ser redefinido por el usuario en una clase derivada para especificar las operaciones que se desean realizar con el nodo visitado. Es invocado por el método <i>inorden</i> .

A continuación se presenta el código correspondiente a la definición de la clase *CArbolBinE*<T>:

```
// CArbolBinE.h - Declaración de la clase CArbolBinE<T>
#ifndef _CARBOLBINE_H_
#define _CARBOLBINE_H_

////////////////////////////////////
// Clase abstracta: árbol binario equilibrado. Para utilizar
// los métodos proporcionados por esta clase, tendremos que
// crear una clase derivada de ella y redefinir los métodos:
// leerdatos, comparar y procesar.
//
template<class T>
class CArbolBinE
{
protected:
    // Nodo de un árbol binario
    class CNodo
    {
    public:
        int numeroDeNodos; // nodos del subárbol que tiene esta raíz
        T datos;           // define el área de datos
        CNodo *izquierdo; // raíz del subárbol izquierdo
        CNodo *derecho;   // raíz del subárbol derecho
    public:
        CNodo() { numeroDeNodos = 0; izquierdo = derecho = 0; }
    };
    CNodo *raiz; // raíz del árbol
    CNodo *construirArbol(int n); // método recursivo
};
```

```

void buscar(T& obj, CNodo *r, T *& datos, int& pos, int& i);
T *obtenerNodo(int nodo_i, CNodo *raiz);

public:
CARbolBinE() { raiz = 0; } // constructor
virtual ~CARbolBinE(); // destructor
void borrarArbol(CNodo *r); // borra todo el árbol

// El método siguiente debe ser redefinido en la clase derivada
// para que permita leer los datos a los que hace referencia un
// nodo del árbol. Devuelve el objeto de datos.
virtual T leerDatos() = 0;

// El método siguiente debe ser redefinido en una clase
// derivada para que permita comparar dos nodos del árbol
// por el atributo que necesitamos en cada momento.
// Devuelve un valor menor, igual o mayor que cero, dependiendo
// de que obj1 sea menor, igual o mayor que obj2.
virtual int comparar(T& obj1, T& obj2) = 0;

// El método siguiente debe ser redefinido en una clase
// derivada para que permita especificar las operaciones
// que se deseen realizar con el nodo visitado.
virtual void procesar(T& obj) = 0;

void construirArbolEquilibrado(int n);
T *obtenerNodo(int nodo_i);
T *buscar(T& obj, int *pos_ret = 0);
void inorden(CNodo *r = 0, bool nodoRaiz = true);
};
////////////////////////////////////
#include "CARbolBinE.cpp"

#endif // _CARBOLBINE_H_

// CARbolBinE.cpp - Definición de la clase CARbolBinE<T>
//
template<class T>
CARbolBinE<T>::~~CARbolBinE() // destructor
{
    borrarArbol(raiz);
    raiz = 0;
}

template<class T>
void CARbolBinE<T>::borrarArbol(CNodo *r)
{
    if ( r != 0 )
    {
        borrarArbol(r->izquierdo); // se visita el subárbol izquierdo
        borrarArbol(r->derecho); // se visita el subárbol derecho
    }
}

```

```

        delete r;
    }
}

template<class T>
typename CArbolBinE<T>::CNodo *CArbolBinE<T>::construirArbol(int n)
{
    // Construye un árbol de n nodos perfectamente equilibrado
    CNodo *nodo = 0;
    int ni = 0, nd = 0;

    if (n == 0)
        return 0;
    else
    {
        ni = n / 2;        // nodos del subárbol izquierdo
        nd = n - ni - 1; // nodos del subárbol derecho
        nodo = new CNodo();
        nodo->numeroDeNodos = n;
        nodo->izquierdo = construirArbol(ni);
        nodo->datos = leerDatos();
        nodo->derecho = construirArbol(nd);
        return nodo;
    }
}

template<class T>
void CArbolBinE<T>::construirArbolEquilibrado(int n)
{
    raiz = construirArbol(n);
}

template<class T>
T *CArbolBinE<T>::obtenerNodo(int nodo_i)
{
    return obtenerNodo(nodo_i, raiz);
}

template<class T>
T *CArbolBinE<T>::obtenerNodo(int nodo_i, CNodo *r)
{
    // Este método permite devolver los datos del nodo i.
    // Los nodos se consideran numerados (0, 1, 2, ...) según
    // el orden en el que son accedidos con la forma "inorden".
    int ni = 0, nd = 0, n = r->numeroDeNodos;
    if (r == 0 || nodo_i < 0 || nodo_i > n) return 0;

    ni = n / 2;        // nodos del subárbol izquierdo
    nd = n - ni - 1; // nodos del subárbol derecho
    if (nodo_i == ni)
        return &r->datos; // nodo actual (raíz subárbol)
}

```

```

else if ( nodo_i < ni )
    // Subárbol izquierdo
    return obtenerNodo(nodo_i, r->izquierdo);
else
    // Subárbol derecho; ajustar el índice en este subárbol
    // descontando los nodos del subárbol izquierdo y el actual
    return obtenerNodo(nodo_i - ni - 1, r->derecho);
return 0;
}

```

```

template<class T>

```

```

T *CARbolBinE<T>::buscar(T& obj, int *pos_ret)

```

```

{
    T *datos = 0;
    int pos_ini = 0, pos = pos_ret ? *pos_ret : 0;
    buscar(obj, raiz, datos, pos, pos_ini);
    if (pos_ret) *pos_ret = pos;
    return datos;
}

```

```

template<class T>

```

```

void CARbolBinE<T>::buscar(T& obj, CNodeo *r, T *& datos, int& pos, int& i)

```

```

{
    // El método buscar permite acceder a un determinado nodo.
    // Si los datos especificados por "obj" se localizan en el
    // árbol referenciado por "r" a partir de la posición "pos",
    // "buscar" devuelve en "datos" un puntero a esos datos;
    // en otro caso, devuelve 0 (valor inicial).
    // Los nodos se consideran numerados (0, 1, 2, ...) según
    // el orden en el que son accedidos por el método "inorden".
    // "i" es la posición del nodo en proceso.
    // "pos" devuelve la posición del nodo encontrado.
    CNodeo *actual = r;

    if (actual != 0 && datos == 0)
    {
        buscar(obj, actual->izquierdo, datos, pos, i);
        if (datos == 0 && pos-- <= 0)
            // La primera condición que aparece en el if anterior es
            // necesaria para que una vez encontrado el nodo no se
            // decremente "pos" en el camino de retorno por la pila de
            // llamadas.
            if (comparar(obj, actual->datos) == 0)
            {
                datos = &actual->datos; // nodo encontrado
                pos = i;
            }
        i++; // posición del siguiente nodo que será accedido
        buscar(obj, actual->derecho, datos, pos, i);
    }
}

```

```

template<class T>
void CArbolBinE<T>::inorden(CNodo *r, bool nodoRaiz)
{
    // El método recursivo inorden visita los nodos del árbol
    // utilizando la forma inorden; esto es, primero se visita
    // el subárbol izquierdo, después se visita la raíz, y por
    // último, el subárbol derecho.
    // Si el segundo parámetro es true, la visita comienza
    // en la raíz independientemente del primer parámetro.
    CNodo *actual = 0;

    if (nodoRaiz)
        actual = raiz; // partir de la raíz
    else
        actual = r;    // partir de un nodo cualquiera
    if (actual != 0)
    {
        inorden(actual->izquierdo, false); // visitar subárbol izq.
        // Procesar los datos del nodo visitado
        procesar(actual->datos);
        inorden(actual->derecho, false); // visitar subárbol dcho.
    }
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

El proceso de construcción lo lleva a cabo el método protegido recursivo denominado *construirArbol*, el cual construye un árbol de n nodos (este método es invocado por el método público *construirArbolEquilibrado*).

El método *construirArbol* tiene un parámetro entero que se corresponde con el número de nodos del árbol y devuelve un puntero al nodo raíz del árbol construido. En realidad diremos que devuelve un puntero a cada subárbol construido, lo que permite realizar los enlaces entre nodos. Observe que para cada nodo se ejecutan las dos sentencias siguientes:

```

nodo->izquierdo = construirArbol(ni);
nodo->derecho = construirArbol(nd);

```

que asignan a los atributos *izquierdo* y *derecho* de cada nodo los punteros a sus subárboles izquierdo y derecho, respectivamente.

Los datos se almacenan en los nodos según el orden en el que son accedidos con la forma *inorden*. Así, recorriendo el árbol de esta forma, podemos recuperar los datos en el mismo orden en el que los hayamos introducido. Si además introducimos los datos ordenados ascendentemente, el árbol construido sería equilibrado y de búsqueda.

La función *obtenerNodo* permite acceder a un nodo por posición. Esto es, los nodos se consideran numerados (0, 1, 2, ...) según el orden en el que son accedidos con la forma inorden. Utilizando esta forma, el nodo más a la izquierda es el primero accedido y por lo tanto el de índice 0. Según esto, los índices virtuales que estamos asignando a los nodos del árbol cumplen el algoritmo que define un árbol de búsqueda, algoritmo que empleamos por lo tanto para acceder al nodo *i*: todos los índices de los nodos del subárbol izquierdo de un nodo son menores que el índice de este nodo y los del subárbol derecho mayores.

El método *buscar* permite acceder a unos datos determinados, recorriendo el árbol desde el nodo raíz y comenzando la búsqueda desde cualquier nodo. El árbol se recorre en la forma *inorden*. Para facilitar la labor del usuario de la clase, se han implementado dos sobrecargas de este método: una pública con dos parámetros (el último opcional) y otra protegida con cinco. El método público invoca al protegido asegurando de esta manera que el puntero a los datos buscados, tercer parámetro de este último, valga inicialmente cero, valor que será devuelto si no se encuentra el nodo. El cuarto parámetro indica la posición del nodo a partir del cual se quiere realizar la búsqueda (independientemente de que el árbol se empiece a recorrer desde el nodo raíz); de esta forma se puede buscar un nodo aunque su clave de búsqueda esté repetida. Y el quinto parámetro permite almacenar la posición del nodo actual; de esta forma se puede devolver la posición del nodo encontrado.

```
void CArbolBinE<T>::buscar(T& obj, CNodo *r, T *& datos, int& pos, int& i)
```

Este método es recursivo. Analicemos sus tres últimos parámetros para ver por qué son pasados por referencia: el tercero, para poder devolver al método llamante un puntero al área de datos; el cuarto, para poder devolver la posición del nodo donde están esos datos y el quinto, para que todas las ejecuciones de este método en curso puedan actualizar la posición del nodo actual en proceso. Esto es, este último parámetro se ha definido así para disponer de una región de almacenamiento que pueda ser compartida por las distintas ejecuciones en curso de *buscar* (piense que es un método recursivo y que cada llamada a sí mismo genera una nueva ejecución de dicho método). Con una variable local al método no podríamos evidentemente realizar la función que tiene *i* (por ser local al método, su valor no podría ser alterado por cualquiera de las ejecuciones en curso). Una variable estática, en lugar de una local, sí puede ser modificada por todas las ejecuciones en curso, pero ponerla a 0 al finalizar una búsqueda y antes de iniciar la siguiente complica excesivamente el código. Lo explicado es aplicable también al parámetro *pos*.

Utilización de la clase `CArbolBinE<T>`

La clase `CArbolBinE<T>` es una clase abstracta; por lo tanto, para hacer uso del soporte que proporciona para la construcción y manipulación de árboles binarios perfectamente equilibrados, tendremos que derivar una clase de ella y redefinir los métodos abstractos heredados: *leerDatos*, *comparar* y *procesar*. La redefinición de estos métodos está condicionada a la clase de objetos que formarán parte del árbol.

Como ejemplo, vamos a construir un árbol binario perfectamente equilibrado en el que cada nodo haga referencia a un objeto de la clase `CDatos` ya utilizada anteriormente en este mismo capítulo.

El método *leerDatos* obtendrá los datos *nombre* y *nota*, a partir de ellos construirá un objeto `CDatos` y devolverá el objeto construido para su inserción en el árbol. Los métodos *comparar* y *procesar* se definen igual que en la clase `CArbolBinarioDeBusqueda`.

Según lo expuesto, la clase `CArbolBinarioEquilibrado` derivada de `CArbolBinE<T>` puede ser de la forma siguiente:

```
// CArbolBinarioEquilibrado.h - Declaración de la clase
//                                     CArbolBinarioEquilibrado
#if !defined( _CARBOLBINARIOEQUILIBRADO_H_ )
#define _CARBOLBINARIOEQUILIBRADO_H_
#include "CArbolBinE.h"
#include "datos.h"

////////////////////////////////////
// Clase derivada de la clase abstracta CArbolBinE. Redefine los
// métodos: leerDatos, comparar y procesar.
//
class CArbolBinarioEquilibrado : public CArbolBinE<CDatos>
{
public:
    CDatos leerDatos();
    int comparar(CDatos& obj1, CDatos& obj2);
    void procesar(CDatos& obj);
};
////////////////////////////////////
#endif // _CARBOLBINARIOEQUILIBRADO_H_

// CArbolBinarioEquilibrado.cpp - Definición de la clase
//                                     CArbolBinarioEquilibrado
#include <iostream>
```

```

#include <string>
#include "CArbolBinarioEquilibrado.h"
#include "leerdatos.h"
using namespace std;

////////////////////////////////////
CDatos CArbolBinarioEquilibrado::leerDatos()
{
    string nombre;
    double nota;
    cout << "nombre: "; leerDato(nombre);
    cout << "nota:   "; leerDato(nota);
    return CDatos(nombre, nota);
}

int CArbolBinarioEquilibrado::comparar(CDatos& obj1, CDatos& obj2)
{
    // Permite comparar dos nodos del árbol por el atributo
    // nombre.
    string str1(obj1.obtenerNombre());
    string str2(obj2.obtenerNombre());
    return str1.compare(str2);
}

void CArbolBinarioEquilibrado::procesar(CDatos& obj)
{
    // Permite mostrar los datos del nodo visitado.
    string nombre(obj.obtenerNombre());
    double nota = obj.obtenerNota();
    cout << nombre << " " << nota << endl;
}
////////////////////////////////////

```

Cuando se declare un objeto de la clase *CArbolBinarioEquilibrado*, el constructor de esta clase invoca al constructor *CArbolBinE<T>* de su clase base, que creará un árbol vacío (*raiz = 0*). El atributo *raiz* apunta siempre a la raíz del árbol.

Finalmente, escribiremos una aplicación *Test* que, utilizando la clase *CArbolBinarioEquilibrado*, cree un objeto *arbolbe* correspondiente a un árbol binario perfectamente equilibrado en el que cada nodo haga referencia a un objeto *CDatos*. De forma resumida, la aplicación *Test*:

1. Creará un objeto *arbolbe* de la clase *CArbolBinarioEquilibrado*.
2. Construirá el árbol equilibrado de *n* nodos, enviando al objeto *arbolbe* el mensaje *construirArbolEquilibrado*.
3. Mostrará los datos almacenados en el árbol para comprobar que se creó como esperábamos.

4. Obtendrá el nodo *i* del árbol invocando a la función *obtenerNodo*.
5. Buscará en el árbol todas las ocurrencias de un nombre dado invocando a la función *buscar*.

```
// test.cpp - Crear un árbol binario perfectamente equilibrado
#include <iostream>
#include <string>
#include "CArbolBinarioEquilibrado.h"
#include "datos.h"
#include "leerdatos.h"
using namespace std;

int main()
{
    CArbolBinarioEquilibrado arbolbe;

    int numeroDeNodos, pos = 0;
    cout << "Número de nodos: ";
    leerDato(numeroDeNodos);
    arbolbe.construirArbolEquilibrado(numeroDeNodos);
    cout << endl;

    // Mostrar los nodos del árbol
    cout << "\nArbol:\n";
    arbolbe.inorden();

    // Obtener los datos del nodo i
    cout << "Nodo (0,1,2,...): ";
    cin >> pos; cin.ignore();
    CDatos *pobj = arbolbe.obtenerNodo(pos);
    if ( pobj == 0 )
        cout << "La búsqueda falló\n";
    else
        cout << "Nombre " << pobj->obtenerNombre()
            << ", nota " << pobj->obtenerNota() << endl;

    // Buscar todas las ocurrencias de nombre
    string nombre;
    cout << "nombre a buscar: "; leerDato(nombre);
    CDatos obj(nombre, 0);
    // Buscar la primera ocurrencia a partir de pos
    pos = 0;
    pobj = arbolbe.buscar(obj, &pos);
    if (pobj == 0) cout << "La búsqueda falló\n";
    while (pobj != 0)
    {
        cout << pos << ".- " << pobj->obtenerNombre() << " "
            << pobj->obtenerNota() << '\n';
        // Buscar más ocurrencias con el mismo nombre
    }
}
```

```

    pos++;
    pobj = arbolbe.buscar(obj, &pos);
}

return 0;
}

```

CLASES RELACIONADAS DE LA BIBLIOTECA C++

La biblioteca estándar de C++ proporciona un conjunto de plantillas, conocidas genéricamente como contenedores, entre los que cabe destacar los siguientes:

- `<vector>`. Matriz unidimensional de elementos de tipo *T*.
- `<list>`. Lista doblemente enlazada de elementos de tipo *T*.
- `<deque>`. Cola de doble extremo de elementos de tipo *T*.
- `<queue>`. Cola de elementos de tipo *T*.
- `<stack>`. Pila de elementos de tipo *T*.
- `<map>`. Matriz unidimensional asociativa de elementos de tipo *T*.
- `<set>`. Conjunto de elementos de tipo *T* (contenedor asociativo).
- `<bitset>`. Matriz unidimensional que almacena un conjunto de bits.

Para mostrar cómo se utilizan estas plantillas vamos a realizar una aplicación con **list** igual a la realizada anteriormente con la plantilla *CListaLinealSE*. El resto de las plantillas puede verlas en la documentación sobre la biblioteca de C++ proporcionada en el CD-ROM que acompaña al libro.

Plantilla list

La plantilla de clase **list** está declarada en el fichero de cabecera `<list>`. Tiene unas características similares a nuestra plantilla *CListaLinealSE*, implementada anteriormente en este mismo capítulo, excepto que no incluye la indexación, porque comparándola con **vector** resulta muy lenta. La tabla siguiente muestra algunos de los métodos proporcionados por esta clase:

Método	Significado
<i>list</i>	Es el constructor de la clase. Tiene uno sin parámetros y varios con ellos; por ejemplo, para construir una lista con un número específico de elementos.
<i>size</i>	Devuelve un valor de tipo int correspondiente al número de elementos de la lista. No tiene parámetros.

Método	Significado
<i>insert</i>	Añade un elemento en la posición <i>i</i> . Tiene dos parámetros: un iterador que hace referencia al objeto de la posición <i>i</i> (véase <i>Iteradores</i> en el capítulo 6) y una referencia al objeto a añadir. No devuelve ningún valor.
<i>push_front</i>	Añade un elemento al principio. Tiene un parámetro que es una referencia al objeto a añadir. No devuelve ningún valor.
<i>push_back</i>	Añade un elemento al final. Tiene un parámetro que es una referencia al objeto a añadir. No devuelve ningún valor.
<i>erase</i>	Borra el elemento de la posición <i>i</i> . Tiene un parámetro que es un iterador que hace referencia al objeto de la posición <i>i</i> . Devuelve un iterador al elemento siguiente al borrado o al final de la lista si el elemento no existe.
<i>pop_front</i>	Borra el primer elemento. No tiene parámetros y no devuelve nada.
<i>pop_back</i>	Borra el último elemento. No tiene parámetros y no devuelve nada.
<i>clear</i>	Borra todos los elementos de la lista.
<i>front</i>	Retorna una referencia al primer elemento de la lista.
<i>back</i>	Retorna una referencia al último elemento de la lista.

A continuación se presenta otra versión de la aplicación *Test* realizada para probar la plantilla *CListaLinealSE*, pero utilizando ahora la plantilla de clase **list**:

```
// Test.cpp - Utilización de la plantilla list
#include <iostream>
#include "leerdatos.h" // para leerDato()
#include <list>
#include "datos.h"
using namespace std;

void mostrarLista(list<CDatos>& lse)
{
    // Mostrar todos los elementos de la lista
    list<CDatos>::iterator e;
    int i = 0;
    for (e = lse.begin(); e != lse.end(); ++e)
    {
        cout << i++ << ".- " << (*e).obtenerNombre() << ' '
            << (*e).obtenerNota() << '\n';
    }
}

int main()
{
```

```
// Crear una lista lineal vacía
list<CDatos> lse;
list<CDatos>::iterator e;

// Leer datos y añadirlos a la lista
string nombre;
double nota;
int i = 0;
CDatos datos;
cout << "Introducir datos. Finalizar con Ctrl+Z.\n";
cout << "nombre: ";
while (leerDato(nombre))
{
    cout << "nota:   ";
    leerDato(nota);
    datos = CDatos(nombre, nota);
    lse.push_back(datos);
    cout << "nombre: ";
}

// Añadir un objeto al principio
datos = CDatos("abcd", 10);
lse.push_front(datos);

// Añadir un objeto en la posición 1
datos = CDatos("defg", 9.5);
lse.insert(++lse.begin(), datos);

cout << "\n\n";
// Mostrar el primero
CDatos& obj = lse.front();
cout << "Primero: " << obj.obtenerNombre() << ' '
    << obj.obtenerNota() << '\n';

// Mostrar el último
cout << "Último: " << lse.back().obtenerNombre() << ' '
    << lse.back().obtenerNota() << '\n';

// Mostrar todos
cout << "Lista:\n";
mostrarLista(lse);

// Borrar el elemento de índice 2
for (i = 0, e = lse.begin(); i < 2; ++i, ++e);
if (e != lse.end()) lse.erase(e);

// Modificar el elemento de índice 1
e = ++lse.begin();
(*e).asignarNota(9);

// Mostrar todos
```

```

    cout << "Lista:\n";
    mostrarLista(lse);
}

```

EJERCICIOS PROPUESTOS

1. Realizar una aplicación que permita crear una lista lineal de elementos de cualquier tipo clasificados ascendentemente. La lista vendrá definida por un objeto de una clase abstracta que denominaremos *CListaLinealSEO<T>* (Clase *Lista Lineal Simplemente Enlazada Ordenada*) y cada elemento de la lista será un objeto de la clase siguiente:

```

class CElemento
{
    // Atributos
public:
    T datos;
    CElemento *siguiente; // siguiente elemento
    // Métodos
public:
    CElemento() { siguiente = 0; } // constructor
    CElemento(T& d, CElemento *s) // constructor
    {
        datos = d;
        siguiente = s;
    }
};

```

La clase *CListaLinealSEO<T>* debe incluir los atributos:

```

CElemento *p; // elemento de cabecera
CElemento *elemAnterior; // elemento anterior
CElemento *elemActual; // elemento actual

```

El atributo *elemActual* de la lista apuntará al último elemento accedido y *elemAnterior*, al anterior al actual, excepto cuando el elemento actual sea el primero, en cuyo caso ambos punteros señalarán a ese elemento. También incluirá los métodos:

```

CListaLinealSEO();
~CListaLinealSEO();
bool listaVacia();
virtual int comparar(T& obj1, T& obj2) = 0;
bool buscar(T& obj);
void anyadir(T& obj);
bool borrar(T& obj);
bool borrarPrimero();

```

```
T *obtenerPrimero();  
T *obtenerActual();  
T *obtenerSiguiente();
```

Todos los métodos expuestos, excepto *listaVacía*, deben actualizar los punteros *elemActual* y *elemAnterior*.

El método *comparar* debe ser redefinido por el usuario en una clase derivada para especificar el tipo de comparación que se desea realizar con dos elementos de la lista. Según esto, debe devolver un entero indicando el resultado de la comparación (-1, 0 ó 1 si $obj1 < obj2$, $obj1 == obj2$, o $obj1 > obj2$, respectivamente). Este método es invocado directamente por el método *buscar* e indirectamente por los métodos *anyadir* y *borrar*, que invocan a *buscar*.

El método *listaVacía* devuelve **true** si la lista está vacía y **false** en caso contrario.

El método *buscar* localiza el punto de inserción de un elemento en una lista ordenada y almacena en *elemActual* un puntero al elemento buscado, si existe, o al siguiente, si no existe, y en *elemAnterior* un puntero al elemento anterior. Tiene un parámetro para almacenar una referencia a los datos de tipo *T* que permitirán localizar el elemento en la lista, y devuelve un valor **true** o **false** dependiendo del resultado de la búsqueda.

El método *anyadir* inserta un elemento en la lista en orden ascendente según una clave seleccionada del área de datos. Para localizar el punto de inserción invocará a *buscar*. La inserción se hará entre los elementos apuntados por *elemAnterior* y *elemActual*. Tiene un parámetro que es una referencia a los datos a añadir. No devuelve nada.

El método *borrar* borra un elemento de la lista. Para buscarlo invoca al método *buscar*. Tiene un parámetro para almacenar una referencia a los datos de tipo *T* que permitirán localizar en la lista el elemento que se desea borrar. Si la operación se realiza satisfactoriamente, el método retorna **true** y devuelve en el parámetro del método los datos correspondientes al elemento borrado; si la operación de borrado no se puede realizar entonces retorna un valor **false**.

El método *borrarPrimero* borra el elemento primero de la lista. Devuelve **true** si la operación se realiza con éxito y **false** en caso contrario.

El método *obtenerPrimero* devuelve un puntero al área de datos del elemento primero y *obtenerActual* al área de datos del elemento actual. Ambas devuelven un cero si la lista está vacía.

El método *obtenerSiguiente* devuelve un puntero al área de datos del elemento siguiente al actual o 0 si la lista está vacía o se intenta ir más allá del último elemento.

En la lista que crearemos a partir de la clase anterior vamos a almacenar objetos de la clase *CDatos* implementada anteriormente en este mismo capítulo.

Pero, para utilizar la clase abstracta *CListaLinealSEO<T>* tenemos que derivar de ella otra clase, por ejemplo *CListaLinealSEOrdenada*, que redefina el método virtual *comparar* para que permita comparar dos objetos *CDatos* por el atributo *nombre*.

Finalmente, realizamos una aplicación que utilizando la clase anterior cree una lista lineal simplemente enlazada y ordenada de objetos *CDatos*.

2. Escribir una aplicación para que, utilizando una pila, simule una calculadora capaz de realizar las operaciones de +, -, * y /. La mayoría de las calculadoras aceptan la notación *infija* y unas pocas, la notación *postfija*. En éstas últimas, para sumar 10 y 20 introduciríamos primero 10, después 20 y por último el +. Cuando se introducen los operandos, se colocan en una pila y cuando se introduce el operador, se sacan dos operandos de la pila, se calcula el resultado y se introduce en la pila. La ventaja de la notación *postfija* es que expresiones complejas pueden evaluarse fácilmente sin mucho código. La calculadora del ejemplo propuesto utilizará la notación *postfija*.

De forma resumida, el programa realizará las siguientes operaciones:

- a) Leerá un dato, operando u operador, y lo almacenará en la variable *oper*.
- b) Analizará *oper*; si se trata de un operando, lo meterá en la pila y si se trata de un operador, sacará los dos últimos operandos de la pila, realizará la operación indicada por dicho operador y meterá el resultado en la pila para poder utilizarlo como operando en una posible siguiente operación.

Para realizar esta aplicación utilizaremos las clases *CPila<T>* derivada de *CListaCircularSE<T>*, *CDatos* y la plantilla *leerDato* diseñada en el capítulo 13 para leer datos desde el teclado (ficheros *leerdatos.h* y *leerdatos.cpp*). Como estas clases y plantillas ya han sido implementadas, en este ejercicio nos limitaremos a utilizar los recursos que proporcionan.

El programa completo se muestra a continuación:

```
// test.cpp - Calculadora
#include <iostream>
#include "cpila.h"
```

```
#include "leerdatos.h" // para leerDato()
#include "datos.h"
using namespace std;

////////////////////////////////////
// Calculadora utilizando una pila. Esta aplicación utiliza las
// clases CPila derivada de CListaCircularSE, CDatos y la
// plantilla leerDatos.
//
bool obtenerOperandos(double operando[], CPila<double>& pila)
{
    int t = 0;
    if ((t = pila.tamanyo()) < 2)
    {
        cout << "Error: teclee " << (2 - t) << " operando(s) más\n";
        return false;
    }
    operando[1] = pila.sacarDePila();
    operando[0] = pila.sacarDePila();
    return true;
}

int main()
{
    CPila<double> pila; // pila de operandos
    double operando[] = {0, 0}; // operando 0 y 1
    // oper almacena la entrada realizada desde el teclado
    char oper[20];
    double resultado, operand;

    cout << "Operaciones: + - * /\n\n";
    cout << "Forma de introducir los datos:\n";
    cout << ">primer operando [Entrar]\n";
    cout << ">segundo operando [Entrar]\n";
    cout << ">operador [Entrar]\n\n";
    cout << "Para salir pulse q\n\n";
    do
    {
        cout << "> ";
        leerDato(oper); // leer un operando o un operador
        switch (oper[0]) // verificar el primer carácter
        {
            case '+':
                if (!obtenerOperandos(operando, pila)) break;
                resultado = operando[0] + operando[1];
                cout << resultado << '\n';
                pila.meterEnPila(resultado);
                break;
            case '-':
                if (!obtenerOperandos(operando, pila)) break;
                resultado = operando[0] - operando[1];
```

```

        cout << resultado << '\n';
        pila.meterEnPila(resultado);
        break;
    case '*':
        if (!obtenerOperandos(operando, pila)) break;
        resultado = operando[0] * operando[1];
        cout << resultado << '\n';
        pila.meterEnPila(resultado);
        break;
    case '/':
        if (!obtenerOperandos(operando, pila)) break;
        if (operando[1] == 0)
        {
            cout << "\nError: división por cero\n";
            break;
        }
        resultado = operando[0] / operando[1];
        cout << resultado << '\n';
        pila.meterEnPila(resultado);
        break;
    case 'q':
        // salir
        break;
    default : // es un operando
        operand = atof(oper);
        pila.meterEnPila(operand);
    }
}
while (oper[0] != 'q');
}

```

3. Escribir una aplicación que permita calcular la frecuencia con la que aparecen las palabras en un fichero de texto. La forma de invocar al programa será:

```
palabras fichero_de_texto
```

donde *fichero_de_texto* es el nombre del fichero de texto del cual deseamos obtener la estadística.

El proceso de contabilizar las palabras que aparezcan en el texto de un determinado fichero lo podemos realizar de la forma siguiente:

- a) Se lee la información del fichero y se descompone en palabras, entendiendo por palabra una secuencia de caracteres delimitada por espacios en blanco, tabuladores, signos de puntuación, etc.

- b) Cada palabra deberá insertarse por orden alfabético ascendente junto con un contador que indique su número de apariciones, en el nodo de una estructura en árbol. Esto facilitará la búsqueda.
- c) Una vez construido el árbol de búsqueda, se presentará por pantalla una estadística con el siguiente formato:

```
...
nombre = 1
obtener = 1
palabras = 1
permítame = 1
programa = 1
que = 2
queremos = 1
será = 1
estadística = 1
texto = 2
un = 1
una = 1
```

```
Total palabras: 44
Total palabras diferentes: 35
```

Según lo expuesto, cada nodo del árbol tendrá que hacer referencia a un área de datos que incluya tanto la palabra como el número de veces que apareció en el texto. Estos datos serán los atributos de una clase *CDatos* declarada así:

```
// datos.h - Declaración de la clase CDatos
#ifndef _DATOS_H_
#define _DATOS_H_
#include <string>

class CDatos
{
private:
    std::string palabra;
    int contador;
public:
    CDatos(std::string pal = "", int n = 0); // constructor
    void asignarPalabra(std::string pal);
    std::string obtenerPalabra() const;
    void asignarContador(int n);
    int obtenerContador() const;
};

#endif // _DATOS_H_
```

El árbol de búsqueda que tenemos que construir será un objeto de la clase *CArbolBinarioDeBusqueda* derivada de *CArbolBinB<T>*. Recuerde que la clase *CArbolBinB<T>* fue implementada anteriormente en este mismo capítulo, al hablar de árboles binarios de búsqueda. La razón por la que derivamos una clase de *CArbolBinB<T>* es que esta clase es abstracta, y se diseñó así para obligar al usuario a redefinir sus métodos virtuales puros *comparar*, que deberá devolver un valor *-1, 0* ó *1* especificando la relación de orden que existe entre dos nodos del árbol, y *procesar*, para especificar las operaciones que se deseen realizar con el nodo visitado.

El método *procesar* de la clase *CArbolBinarioDeBusqueda*, además de visualizar la información almacenada en el objeto *CDatos* referenciado por el nodo visitado, contabilizará el número total de palabras del texto procesado y el número total de palabras diferentes (esto es, como si todas hubieran aparecido sólo una vez en el texto). El resto de los métodos ya fueron explicados al hablar de árboles binarios de búsqueda.

Sólo queda construir una aplicación que cree un objeto de la clase *CArbolBinarioDeBusqueda* a partir de las palabras almacenadas en un fichero y presente los resultados pedidos. El código de esta aplicación se va a apoyar en tres funciones: **main**, *leerFichero* y *palabras*.

El método **main** verificará, cuando se ejecute la aplicación, que se haya pasado como parámetro el nombre del fichero de texto, invocará al método *leerFichero* y una vez construido el árbol, lo recorrerá para visualizar los resultados pedidos.

El método *leerFichero* abre el fichero y lo lee línea a línea. Cada línea leída será pasada como argumento al método *palabras* para su descomposición en palabras con el fin de añadirlas al árbol binario de búsqueda que ha sido declarado como un atributo de la clase aplicación. Para descomponer una línea en palabras utilizaremos la función **strtok** de la biblioteca de C (véanse los apéndices).

CAPÍTULO 16

© F.J.Ceballos/RA-MA

ALGORITMOS

En este capítulo vamos a exponer cómo resolver algunos problemas muy comunes en programación. El primero que nos vamos a plantear es la *ordenación* de objetos en general; la ordenación es tan común que no necesita explicación; algo tan cotidiano como una guía telefónica es un ejemplo de una lista ordenada. Y el segundo problema que vamos a abordar es la búsqueda de objetos en un conjunto; el localizar un determinado teléfono exige una *búsqueda* por algún método. Finalmente veremos los algoritmos que nos proporciona la biblioteca de C++ para dar solución a estos problemas tan comunes a los que nos hemos referido.

ORDENACIÓN DE DATOS

Uno de los procedimientos más comunes y útiles en el procesamiento de datos es la ordenación de los mismos. Se considera ordenar al proceso de reorganizar un conjunto dado de objetos en una secuencia determinada. El objetivo de este proceso generalmente es facilitar la búsqueda de uno o más elementos pertenecientes a un conjunto. Son ejemplos de datos ordenados las listas de los alumnos matriculados en una cierta asignatura, las listas del censo, los índices alfabéticos de los libros, las guías telefónicas, etc. Esto quiere decir que muchos problemas están relacionados de alguna forma con el proceso de ordenación. Es por lo que la ordenación es un problema importante a considerar.

La ordenación, tanto numérica como alfanumérica, sigue las mismas reglas que empleamos nosotros en la vida normal. Esto es, un dato numérico es mayor que otro cuando su valor es más grande, y una cadena de caracteres es mayor que otra cuando está después por orden alfabético.

Podemos agrupar los métodos de ordenación en dos categorías: ordenación de matrices u ordenación interna (cuando los datos se guardan en memoria interna) y

ordenación de ficheros u ordenación externa (cuando los datos se guardan en memoria externa; generalmente en discos).

En este apartado no se trata de analizar exhaustivamente todos los métodos de ordenación y ver sus prestaciones de eficiencia, rapidez, etc. sino que simplemente analizamos desde el punto de vista práctico los métodos más comunes para ordenación de matrices y de ficheros.

Método de la burbuja

Hay muchas formas de ordenar datos, pero una de las más conocidas es la ordenación por el método de la burbuja.

Veamos a continuación el algoritmo correspondiente a este método para ordenar una lista de menor a mayor, partiendo de que los datos a ordenar están almacenados en una matriz de n elementos:

1. Comparamos el primer elemento con el segundo, el segundo con el tercero, el tercero con el cuarto, etc. Cuando el resultado de una comparación sea “mayor que”, se intercambian los valores de los elementos comparados. Con esto conseguimos llevar el valor mayor a la posición n .
2. Repetimos el punto 1, ahora para los $n-1$ primeros elementos de la lista. Con esto conseguimos llevar el valor mayor de éstos a la posición $n-1$.
3. Repetimos el punto 1, ahora para los $n-2$ primeros elementos de la lista y así sucesivamente.
4. La ordenación estará realizada cuando al repetir el *iésimo* proceso de comparación no haya habido ningún intercambio o, en el peor de los casos, después de repetir el proceso de comparación descrito $n-1$ veces.

El pseudocódigo para este algoritmo puede ser el siguiente:

```
<método ordenar(matriz "a" de "n" elementos)>
["a" es un matriz cuyos elementos son  $a_0, a_1, \dots, a_{n-1}$ ]
   $n = n-1$ 
  DO WHILE ("a" no esté ordenado y  $n > 0$  )
     $i = 1$ 
    DO WHILE (  $i \leq n$  )
      IF (  $a[i-1] > a[i]$  ) THEN
        permutar  $a[i-1]$  con  $a[i]$ 
      ENDIF
       $i = i+1$ 
    ENDDO
```



```

    n = n-1
  ENDDO
END <método ordenar>

```

La clase siguiente incluye el método *ordenar_b* que utiliza este algoritmo para ordenar una matriz de tipo *T*, siempre que este tipo defina los operadores = y >.

```

// algoritmos.h - Declaración de la clase CAlgoritmo
#ifndef _ALGORITMO_H_
#define _ALGORITMO_H_

template<class T>
class CAlgoritmo
{
public:
    static void ordenar_b(T m[], int n); // burbuja
};

#include "algoritmos.cpp"
#endif // _ALGORITMO_H_

// algoritmos.cpp - Definición de la clase CAlgoritmo
#include "algoritmos.h"

////////////////////////////////////
// Ordenar ascendentemente objetos de tipo T por el método de la
// burbuja, siempre que esta clase de objetos defina los operadores
// = y >.
template<class T>
void CAlgoritmo<T>::ordenar_b(T m[], int numero_de_elementos)
{
    T aux;
    int i;
    bool s = true;
    while (s && (--numero_de_elementos > 0))
    {
        s = false; // no permutación
        for (i = 1; i <= numero_de_elementos; i++)
            // ¿ el elemento (i-1) es mayor que el (i) ?
            if (m[i-1] > m[i])
            {
                // permutar los elementos (i-1) e (i)
                aux = m[i-1];
                m[i-1] = m[i];
                m[i] = aux;
                s = true; // permutación
            }
    }
}
////////////////////////////////////

```

Observe que *s* inicialmente vale **false** para cada iteración y toma el valor **true** cuando al menos se efectúa un cambio entre dos elementos. Si en una exploración a lo largo de la lista no se efectúa cambio alguno, *s* permanecerá valiendo **false**, lo que indica que la lista está ordenada, terminando así el proceso.

Cuando se analiza un método de ordenación, hay que determinar cuántas comparaciones e intercambios se realizan para el caso más favorable, para el caso medio y para el caso más desfavorable.

En el método de la burbuja se realizan $(n-1)(n/2)=(n^2-n)/2$ comparaciones en el caso más desfavorable, donde *n* es el número de elementos a ordenar. Para el caso más favorable (la lista está ordenada), el número de intercambios es 0. Para el caso medio es $3(n^2-n)/4$, hay tres intercambios por cada elemento desordenado. Y para el caso menos favorable, el número de intercambios es $3(n^2-n)/2$. El análisis matemático que conduce a estos valores queda fuera del propósito de este libro. El tiempo de ejecución es del orden de n^2 y está directamente relacionado con el número de comparaciones y de intercambios.

La siguiente aplicación ordena una matriz **double**, otra de tipo **string** y otra de tipo *CDatos* utilizando el método *ordenar_b* de la clase *CAgoritmo<T>*. La clase *CDatos* es la que hemos venido utilizando en el capítulo anterior (*Estructuras dinámicas*) y define los operadores ==, < y >, y por omisión el de asignación.

```
// test.cpp - Ordenación y búsqueda
#include <iostream>
#include <string>
#include "algoritmos.h"
#include "datos.h"
using namespace std;

int main()
{
    // Burbuja:
    // Matriz numérica
    double m[] = {3,2,1,5,4};
    CAgoritmo<double>::ordenar_b(m, 5);
    for (int i = 0; i < 5; i++)
        cout << m[i] << " ";
    cout << endl;

    // Matriz de cadenas de caracteres
    string s[] = {"ccc","bbb","aaa","eee","ddd"};
    CAgoritmo<string>::ordenar_b(s, 5);
    for (int i = 0; i < 5; i++)
        cout << s[i] << " ";
    cout << endl;
}
```

```

// Matriz de objetos CDatos (almacenan nombre y nota)
CDatos a[3];
a[0] = CDatos("bbb", 10);
a[1] = CDatos("ccc", 9.5);
a[2] = CDatos("aaa", 9);
CAlgoritmo<CDatos>::ordenar_b(a, 3);
for (int i = 0; i < 3; i++)
    cout << a[i].obtenerNombre() << " "
         << a[i].obtenerNota() << '\n';
cout << endl;
}

```

Método de inserción

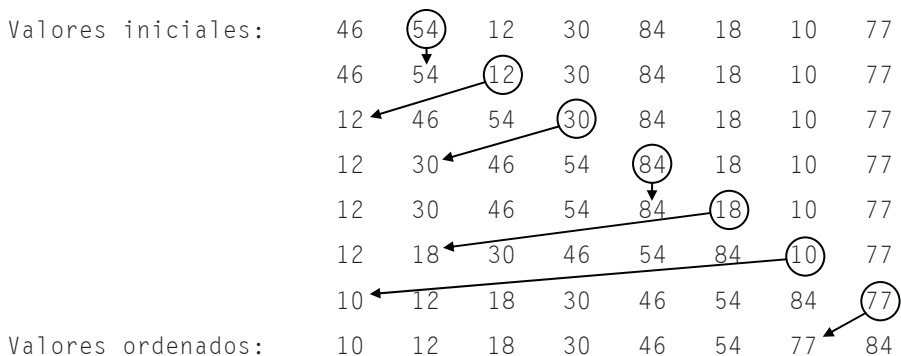
El algoritmo para este método de ordenación es el siguiente: inicialmente, se ordenan los dos primeros elementos de la matriz, luego se inserta el tercer elemento en la posición correcta con respecto a los dos primeros, a continuación se inserta el cuarto elemento en la posición correcta con respecto a los tres primeros elementos ya ordenados y así sucesivamente hasta llegar al último elemento de la matriz. El pseudocódigo para este algoritmo puede ser el siguiente:

```

<método inserción(matriz "a" de "n" elementos)>
["a" es un matriz cuyos elementos son  $a_0, a_1, \dots, a_{n-1}$ ]
  i = 1
  DO WHILE ( i < n )
    x = a[i]
    insertar x en la posición correcta entre  $a_0$  y  $a_i$ 
  ENDDO
END <inserción>

```

La figura siguiente muestra el resultado que se obtiene al aplicar este algoritmo sobre una lista de números:



La plantilla de método *ordenar_i* que se muestra a continuación implementa este algoritmo para ordenar una matriz de tipo *T*, siempre que este tipo defina los operadores $=$ y $<$.

```
template<class T>
void CAlgoritmo<T>::ordenar_i(T m[], int numero_de_elementos)
{
    T x;
    int i, k;
    // Desde el segundo elemento
    for (i = 1; i < numero_de_elementos; i++)
    {
        x = m[i];
        k = i-1;
        // Para k=-1, se ha alcanzado el extremo izquierdo.
        while (k >=0 && x < m[k])
        {
            m[k+1] = m[k]; // hacer hueco para insertar
            k--;
        }
        m[k+1] = x; // insertar x en su lugar
    }
}
```

Análisis del método de inserción directa:

	comparaciones	intercambios
<i>caso más favorable</i>	$n-1$	$2(n-1)$
<i>caso medio</i>	$(n^2 + n - 2)/4$	$(n^2 + 9n - 10)/4$
<i>caso menos favorable</i>	$(n^2 + n)/2 - 1$	$(n^2 + 3n - 4)/2$

Para el método de inserción, el tiempo de ejecución es función de n^2 y está directamente relacionado con el número de comparaciones y de intercambios.

Método quicksort

El método de ordenación *quicksort* está generalmente considerado como el mejor algoritmo de ordenación disponible actualmente. El proceso seguido por este algoritmo es el siguiente:

1. Se selecciona un valor perteneciente al rango de valores de la matriz. Este valor se puede escoger aleatoriamente o haciendo la media de un pequeño conjunto de valores tomados de la matriz. El valor óptimo sería la mediana (el valor que es menor o igual que los valores correspondientes a la mitad de los elementos de la matriz y mayor o igual que los valores correspondientes a la

otra mitad). No obstante, incluso en el peor de los casos (el valor escogido está en un extremo), *quicksort* funciona correctamente.

2. Se divide la matriz en dos partes: una con todos los elementos menores que el valor seleccionado y otra con todos los elementos mayores o iguales.
3. Se repiten los puntos 1 y 2 para cada una de las partes en las que se ha dividido la matriz, hasta que esté ordenada.

El proceso descrito es esencialmente recursivo. Según lo expuesto, el pseudocódigo para este algoritmo puede ser el siguiente:

```

<método qs(matriz "a")>
  Se elige un valor x de la matriz
  DO WHILE ( "a" no esté dividido en dos partes )
    [dividir "a" en dos partes: a_inf y a_sup]
    a_inf con los elementos a_i < x
    a_sup con los elementos a_i >= x
  ENDDO
  IF ( existe a_inf ) THEN
    qs( a_inf )
  ENDIF
  IF ( existe a_sup ) THEN
    qs( a_sup )
  ENDIF
END <qs>

```

A continuación se muestra una versión de este algoritmo, que selecciona el elemento medio de la matriz para proceder a dividirla en dos partes. Esto resulta fácil de implementar, aunque no siempre da lugar a una buena elección. A pesar de ello, funciona correctamente.

```

template<class T>
void CAlgoritmo<T>::ordenar_qs(T m[], int numero_de_elementos)
{
  qs(m, 0, numero_de_elementos - 1);
}

// Método recursivo qs
template<class T>
void CAlgoritmo<T>::qs(T m[], int inf, int sup)
{
  T mitad, x;
  int izq, der;
  izq = inf; der = sup;
  mitad = m[(izq + der) / 2];
  do
  {

```

```

while (m[izq] < mitad && izq < sup) izq++;
while (mitad < m[der] && der > inf) der--;
if (izq <= der)
{
    x = m[izq]; m[izq] = m[der]; m[der] = x;
    izq++; der--;
}
}
while (izq <= der);
if (inf < der) qs(m, inf, der);
if (izq < sup) qs(m, izq, sup);
}

```

Observamos que cuando el valor *mitad* se corresponde con uno de los valores de la lista, las condiciones $izq < sup$ y $der > inf$ de las sentencias

```

while (m[izq] < mitad && izq < sup) izq++;
while (mitad < m[der] && der > inf) der--;

```

no serían necesarias. En cambio, si el valor *mitad* es un valor no coincidente con un elemento de la lista, pero que está dentro del rango de valores al que pertenecen los elementos de la misma, esas condiciones son necesarias para evitar que se puedan sobrepasar los límites de los índices de la matriz.

Para experimentarlo, pruebe como ejemplo la lista de valores *1 1 3 1 1* y elija *mitad = 2* fijo.

En el método *quicksort*, en el caso más favorable, esto es, cada vez se selecciona la mediana obteniéndose dos particiones iguales, se realizan $n \times \log n$ comparaciones y $n/6 \times \log n$ intercambios, donde n es el número de elementos a ordenar; en el caso medio, el rendimiento es inferior al caso óptimo en un factor de $2 \times \log 2$; y en el caso menos favorable, esto es, cada vez se selecciona el valor mayor obteniéndose una partición de $n-1$ elementos y otra de un elemento, el rendimiento es del orden de $n \times n = n^2$. Con el fin de mejorar el caso menos favorable, se sugiere elegir, cada vez, un valor aleatoriamente o un valor que sea la mediana de un pequeño conjunto de valores tomados de la matriz.

Comparación de los métodos expuestos

Si medimos los tiempos consumidos por los métodos de ordenación estudiados anteriormente, observaremos que el método de la burbuja es el peor de los métodos; el método de inserción directa mejora considerablemente y el método *quicksort* es el más rápido y mejor método de ordenación de matrices con diferencia.

BÚSQUEDA DE DATOS

El objetivo de ordenar un conjunto de objetos es, generalmente, facilitar la búsqueda de uno o más elementos pertenecientes a ese conjunto. Es posible realizar dicha búsqueda sin que el conjunto de objetos esté ordenado, pero esto trae como consecuencia un mayor tiempo de proceso.

Búsqueda secuencial

Este método de búsqueda, aunque válido, es el menos eficiente. Se basa en comparar el valor que se desea buscar con cada uno de los valores de la matriz. La matriz no tiene por qué estar ordenada.

El pseudocódigo para este método de búsqueda puede ser el siguiente:

```
<método búsqueda_S( matriz a, valor que queremos buscar)>
  i = 0
  DO WHILE ( no encontrado )
    IF ( valor = a[i] )
      encontrado
    ENDIF
    i = i+1
  ENDDO
END <búsqueda_S>
```

Como ejercicio, escribir el código correspondiente a un método que permita buscar un valor, previamente leído, en un matriz.

Búsqueda binaria

Un método eficiente de búsqueda, que puede aplicarse a las matrices clasificadas, es la *búsqueda binaria*. Si partimos de que los elementos de la matriz están almacenados en orden ascendente, el proceso de búsqueda binaria puede describirse así: se selecciona el elemento del centro o aproximadamente del centro de la matriz. Si el valor a buscar no coincide con el elemento seleccionado y es mayor que él, se continúa la búsqueda en la segunda mitad de la matriz. Si, por el contrario, el valor a buscar es menor que el valor del elemento seleccionado, la búsqueda continúa en la primera mitad de la matriz. En ambos casos, se halla de nuevo el elemento central, correspondiente al nuevo intervalo de búsqueda, repitiéndose el ciclo. El proceso se repite hasta que se encuentra el valor a buscar o bien hasta que el intervalo de búsqueda sea nulo, lo que querrá decir que el elemento buscado no figura en la matriz.

El pseudocódigo para este algoritmo puede ser el siguiente:

```
<método búsquedaBin( matriz a, valor que queremos buscar )>
DO WHILE ( no encontrado y exista un intervalo donde buscar )
  x = elemento mitad del intervalo de búsqueda
  IF ( valor > x ) THEN
    buscar "valor" en la segunda mitad del intervalo de búsqueda
  ELSE
    buscar "valor" en la primera mitad del intervalo de búsqueda
  ENDIF
ENDDO
IF ( se encontró valor ) THEN
  retornar su índice
ELSE
  retornar -1
ENDIF
END <búsquedaBin>
```

A continuación se muestra el código correspondiente a este método.

```
template<class T>
int CAlgoritmo<T>::busquedaBin(T m[], T v, int numero_de_elementos)
{
  // El método busquedaBin devuelve como resultado la posición
  // del valor. Si el valor no se localiza devuelve -1.

  if (numero_de_elementos == 0) return -1;
  int mitad, inf = 0, sup = numero_de_elementos - 1;

  do
  {
    mitad = (inf + sup) / 2;
    if (v > m[mitad])
      inf = mitad + 1;
    else
      sup = mitad - 1;
  }
  while ( m[mitad] != v && inf <= sup);

  if (m[mitad] == v)
    return mitad;
  else
    return -1;
}
```

Búsqueda de cadenas

Uno de los métodos más eficientes en la búsqueda de cadenas dentro de un texto es el algoritmo *Boyer y Moore*. La implementación básica de este método construye una tabla *delta* que se utilizará en la toma de decisiones durante la búsqueda

de una subcadena. Dicha tabla contiene un número de entradas igual al número de caracteres del código que se esté utilizando. Por ejemplo, si se está utilizando el código de caracteres ASCII, la tabla será de 256 entradas. Cada entrada contiene el valor *delta* asociado con el carácter que representa. Por ejemplo, el valor *delta* asociado con *A* estará en la entrada 65 y el valor *delta* asociado con el *espacio en blanco*, en la entrada 32. El valor *delta* para un carácter es la posición de la ocurrencia más a la derecha de ese carácter respecto a la posición final en la cadena buscada. Las entradas correspondientes a los caracteres que no pertenecen a la cadena a buscar tienen un valor igual a la longitud de esta cadena.

Por lo tanto, para definir la tabla *delta* para una determinada subcadena a buscar, construimos una matriz con todos sus elementos iniciados a la longitud de dicha cadena y, luego, asignamos el valor *delta* para cada carácter de la subcadena, así:

```
for ( i = 0; i < longitud_cadena_patron; i++ )
    delta[cadena_patron[i]] = longitud_cadena_patron - i - 1;
```

En el algoritmo de *Boyer y Moore* la comparación se realiza de derecha a izquierda, empezando desde el principio del texto. Es decir, se empieza comparando el último carácter de la cadena que se busca con el correspondiente carácter en el texto donde se busca; si los caracteres no coinciden, la cadena que se busca se desplaza hacia la derecha un número de caracteres igual al valor indicado por la entrada en la tabla *delta* correspondiente al carácter del *texto* que no coincide. Si el carácter no aparece en la cadena que se busca, su valor *delta* es la longitud de la cadena que se busca.

Veamos un ejemplo. Suponga que se desea buscar la cadena “cien” en el texto “Más vale un ya que cien después se hará”. La búsqueda comienza así:

```
Texto:           Más vale un ya que cien después se hará
Cadena a buscar: cien
```

El funcionamiento del algoritmo puede comprenderse mejor situando la cadena a buscar paralela al texto. La comparación es de derecha a izquierda; por lo tanto, se compara el último carácter en la cadena a buscar (*n*) con el carácter que está justamente encima en el texto (*espacio*). Como *n* es distinto de *espacio*, la cadena que se busca debe desplazarse a la derecha un número de caracteres igual al valor indicado por la entrada en la tabla *delta* que corresponde al carácter del *texto* que no coincide. Para la cadena “cien”,

```
delta['c'] = 3
delta['i'] = 2
delta['e'] = 1
delta['n'] = 0
```

El resto de las entradas valen 4 (longitud de la cadena). Según esto, la cadena que se busca se desplaza cuatro posiciones a la derecha (el espacio en blanco no aparece en la cadena que se busca).

Texto: Más vale un ya que cien después se hará
Cadena a buscar: cien

Ahora, n no coincide con e ; luego la cadena se desplaza una posición a la derecha (e tiene un valor asociado de 1).

Texto: Más vale un ya que cien después se hará
Cadena a buscar: cien

n no coincide con *espacio*; se desplaza la cadena cuatro posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
Cadena a buscar: cien

n no coincide con y ; se desplaza la cadena cuatro posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
Cadena a buscar: cien

n no coincide con u ; se desplaza la cadena cuatro posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
Cadena a buscar: cien

n no coincide con i ; se desplaza la cadena dos posiciones a la derecha.

Texto: Más vale un ya que cien después se hará
Cadena a buscar: cien

Todos los caracteres de la cadena coinciden con los correspondientes caracteres en el texto. Para encontrar la cadena se han necesitado sólo $7+3$ comparaciones (7 hasta que se dio la coincidencia del carácter n de “cien” más 3 para verificar que coincidían los tres caracteres restantes). El algoritmo directo habría realizado $20+3$ comparaciones, que en el peor de los casos, serían $i \times longCadBuscar$, donde i es la posición más a la izquierda de la primera ocurrencia de la cadena a buscar en el texto (20 en el ejemplo anterior, suponiendo que la primera posición es la 1) y $longCadBuscar$ es la longitud de la cadena a buscar (4 en el ejemplo anterior). En cambio, el algoritmo *Boyer y Moore* emplearía $k \times (i + longCadBuscar)$ comparaciones, donde $k < 1$.

El algoritmo *Boyer y Moore* es más rápido porque tiene información sobre la cadena que se busca, en la tabla *delta*. El carácter que ha causado la no coincidencia en el texto indica cómo mover la cadena respecto del texto. Si el carácter no coincidente en el texto no existe en la cadena, ésta puede moverse sin problemas a la derecha un número de caracteres igual a su longitud, pues es un gasto de tiempo comparar la cadena con un carácter que ella no contiene. Cuando el carácter no coincidente en el texto está presente en la cadena, el valor *delta* para ese carácter alinea la ocurrencia más a la derecha de ese carácter en la cadena con el carácter en el texto.

A continuación se muestra el código correspondiente al algoritmo *Boyer y Moore*. El método *buscarCadena* es el que realiza el proceso descrito. Este método devuelve la posición de la cadena en el texto o -1 si la cadena no se encuentra (la primera posición es la 0).

```
template<>
int CAlgoritmo<char>::buscarCadena(char texto[], char cadena[])
{
    // Construir la tabla "delta"
    int delta[256];
    int i, longCad = strlen(cadena);
    // Iniciar la tabla "delta"
    for (i = 0; i < 256; i++)
        delta[i] = longCad;
    // Asignar valores a la tabla
    for (i = 0; i < longCad; ++i)
        delta[cadena[i]] = longCad - i - 1;

    // Algoritmo Boyer-Moore
    int j, longTex = strlen(texto);
    i = longCad - 1; // i es el índice dentro del texto
    while (i < longTex)
    {
        j = longCad - 1; // índice dentro de la cadena a buscar
        // Mientras haya coincidencia de caracteres
        while (cadena[j] == texto[i])
        {
            if ( j > 0 )
            {
                // Siguiete posición a la izquierda
                j--; i--;
            }
            else
            {
                // Se llegó al principio de la cadena, luego se encontró.
                return i;
            }
        }
    }
}
```

```
    // Los caracteres no coinciden. Mover i lo que indique el
    // valor "delta" del carácter del texto que no coincide
    i += delta[texto[i]];
}
return -1;
}
```

CLASES RELACIONADAS DE LA BIBLIOTECA C++

La biblioteca estándar de C++ proporciona un conjunto de plantillas, conocidas genéricamente como algoritmos, que pueden aplicarse a cualquier secuencia de cualquier tipo de elementos. Las declaraciones de estos algoritmos generales se localizan en el fichero de cabecera *<algorithm>*. A continuación se expone una descripción breve de los mismos:

- *for_each*. Permite realizar una operación sobre cada uno de los elementos de una secuencia.
- *find*. Encuentra la primera ocurrencia de un valor en una secuencia.
- *count*. Cuenta las ocurrencias de un valor en una secuencia.
- *equal*. Devuelve **true** si dos secuencias son iguales elemento a elemento.
- *search*. Encuentra la primera ocurrencia de una subsecuencia en una secuencia.
- *search_n*. Encuentra la n-ésima ocurrencia de un valor en una secuencia.
- *find_end*. Encuentra la última ocurrencia de una subsecuencia en una secuencia.
- *transform*. Aplica una operación a cada elemento de una secuencia.
- *copy*. Copia una secuencia empezando por el primer elemento.
- *copy_backward*. Copia una secuencia empezando por el último elemento.
- *swap*. Intercambia dos elementos.
- *replace*. Reemplaza elementos con un valor dado.
- *fill*. Reemplaza cada elemento con un valor dado.
- *fill_n*. Reemplaza los primeros *n* elementos con un valor dado.
- *remove*. Elimina elementos con un valor dado.
- *unique*. Elimina elementos adyacentes que son iguales.
- *reverse*. Invierte el orden de los elementos.
- *sort*. Ordena los elementos de una secuencia.
- *binary_search*. Indica si un elemento está en una secuencia ordenada.
- *lower_bound*. Encuentra la primera ocurrencia de un valor en una secuencia ordenada.
- *merge*. Mezcla dos secuencias.
- *min*. Mínimo de dos valores.
- *max*. Máximo de dos valores.

Modo de empleo de los algoritmos

A continuación se muestra un ejemplo de cómo utilizar algunos de los algoritmos de ordenación y búsqueda para que pueda compararlos con los explicados anteriormente en este capítulo:

```
// test.cpp - Ordenación y búsqueda
#include <iostream>
#include <vector>
#include <algorithm>
#include "datos.h"
using namespace std;

void mostrar(CDatos& obj)
{
    cout << obj.obtenerNombre() << " "
         << obj.obtenerNota() << '\n';
}

int main()
{
    // Matriz de objetos CDatos (almacenan nombre y nota).
    // CDatos debe definir el operador < para sort y el operador ==
    // para remove.

    vector<CDatos> a(3);
    a[0] = CDatos("bbb", 10);
    a[1] = CDatos("ccc", 9.5);
    a[2] = CDatos("aaa", 9);

    vector<CDatos> b(3), c(6);
    b[0] = CDatos("fff", 8.5);
    b[1] = CDatos("ddd", 8);
    b[2] = CDatos("eee", 7.5);

    // Fusionar los vectores a y b en c
    merge(a.begin(), a.end(), b.begin(), b.end(), c.begin());

    cout << "Ordenación descendente:\n";
    sort(c.begin(), c.end(), greater<CDatos>());
    for_each(c.begin(), c.end(), mostrar);
    cout << endl;

    cout << "Ordenación ascendente:\n";
    sort(c.begin(), c.end());
    for_each(c.begin(), c.end(), mostrar);
    cout << endl;
}
```

```
// Buscar un elemento en una matriz ordenada: búsqueda binaria.
// La matriz tiene que estar ordenada ascendente.
CDatos x("ccc");
vector<CDatos>::iterator nuevo_end;
if (binary_search(c.begin(), c.end(), x))
    nuevo_end = remove(c.begin(), c.end(), x);
else
    cout << "elemento no encontrado\n\n";
for_each(c.begin(), c.end(), mostrar);
cout << endl;
// La operación remove desplaza los elementos de la secuencia,
// para cubrir el hueco del elemento borrado, sin cambiar su
// tamaño (el último elemento se repite). Para cambiar el tamaño
// de la secuencia utilizar erase:
c.erase (nuevo_end, c.end());
for_each(c.begin(), c.end(), mostrar);
cout << endl;
}
```

EJERCICIOS PROPUESTOS

1. El acceso aleatorio a un fichero permite ordenar la información contenida en el mismo sin tener que copiarla sobre otro fichero, para lo cual aplicaremos un proceso análogo al aplicado a las matrices, lo que simplifica enormemente el proceso ordenación. Esto quiere decir que los métodos expuestos para ordenar matrices pueden ser aplicados también para ordenar ficheros utilizando el acceso aleatorio.

Como ejercicio, añade a la clase *CAgoritmo* implementada anteriormente en este mismo capítulo un método **static** denominado *ordenarFichero* para ordenar un fichero en el que cada registro esté formado por los campos: *nombre* y *nota*. La ordenación del fichero se realizará por el campo *nombre*, de tipo alfabético, empleando el método *quicksort* explicado anteriormente en este mismo capítulo.

P A R T E

4

Apéndices

- Novedades en C++0x
- La biblioteca estándar de C++
- La biblioteca de C
- Entornos de desarrollo
- Instalación del paquete de desarrollo
- Códigos de caracteres

NOVEDADES DE C++0x

C++0x es el nombre de trabajo para el nuevo estándar del lenguaje de programación C++ que reemplazará al estándar ISO/IEC 14882 actual, publicado en 1998 (C++98) y actualizado 2003 (C++03), y que de ser aprobado a lo largo de este año 2009 pasará a llamarse C++09. Es 100% compatible con C++03. Las modificaciones introducidas afectan tanto a la biblioteca estándar como al lenguaje. Entre las nuevas características que se incluirán en este nuevo estándar destacamos las siguientes:

- Cambios en la biblioteca estándar independientes del lenguaje: por ejemplo, plantillas con un número variable de argumentos (*variadic*) y **constexpr**.
- Facilidades para escribir código: **auto**, **enum class**, **long long**, **nullptr**, ángulos derechos (>>) en plantillas o **static_assert**.
- Ayudas para actualizar y mejorar la biblioteca estándar: **constexpr**, listas de iniciadores generales y uniformes, referencias *rvalue*, plantillas *variadic* y una versión de la biblioteca estándar con todas estas características.
- Características relacionadas con la concurrencia: modelo de memoria multithread, **thread_local** o una biblioteca para realizar programación concurrente (hilos).
- Características relacionadas con conceptos: **concepts** (mecanismo para la descripción de los requisitos sobre los tipos y las combinaciones de los mismos lo que mejorará la calidad de los mensajes de error del compilador), sentencia **for** para iterar sobre un conjunto de valores y conceptos en la biblioteca estándar.
- Expresiones *lambda*.

La finalidad de todas estas nuevas características de C++ es mejorar el rendimiento de las aplicaciones durante su construcción y durante su ejecución, mejo-

rar la usabilidad y funcionalidad del lenguaje y proporcionar una biblioteca estándar más completa y segura.

INFERENCIA DE TIPOS

La inferencia de tipos asigna automáticamente un tipo de datos a una variable a partir de una expresión. Para ello, la variable es calificada **auto**. Esto es, el especificador **auto** es un marcador de posición para un tipo que se deduce de una determinada expresión:

```
auto var = expresión;
```

Por ejemplo, en la siguiente sentencia x tendrá el tipo **int** porque es el tipo de su valor de iniciación:

```
auto x = 15;
```

El uso de **auto** es tanto más útil cuanto más difícil sea conocer exactamente el tipo de una variable o expresión. Por ejemplo, considere la siguiente función genérica:

```
template<class T> void mostrarVector(const vector<T>& v)
{
    for (auto p = v.begin(); p != v.end(); ++p)
        cout << *p << "\n";
}
```

¿Cuál es tipo de p ? En este caso **auto** está reemplazando al tipo:

```
typename vector<T>::const_iterator
```

El antiguo significado de **auto** (variable local automática) es redundante y ya no es utilizado.

ÁNGULOS DERECHOS EN EL USO DE PLANTILLAS

Considere el siguiente ejemplo escrito en C++03:

```
var vector<vector<double>>> v1;
```

Observamos que en C++03 era necesario dejar un espacio entre los ángulos sombreados. Ahora, en C++0x ya no es necesario:

```
var vector<vector<double>>> v1;
```

SENTENCIA **for** APLICADA A COLECCIONES

Es posible acceder a cada uno de los elementos de una colección utilizando la siguiente sentencia **for**:

```
for (auto var : colección)
```

Por ejemplo, la siguiente plantilla de función utiliza esta sentencia, primero para multiplicar por 2 cada uno de los elementos del vector pasado como argumento y después, para mostrar cada uno de los elementos del vector:

```
template<class T> void mostrarVector(const vector<T>& v)
{
    for(auto& x : v)
    {
        x *= 2;
    }

    for (auto x : v)
        cout << x << "\n";
}
```

LISTA DE INICIACIÓN

C++0x extiende el lenguaje para que las listas de iniciación que ya utilizábamos cuando definíamos una estructura o una matriz puedan utilizarse ahora también para iniciar otros objetos. Una lista de iniciación puede utilizarse en los siguientes casos, entre otros:

- Para iniciar una variable:

```
int x = {0};
vector<double> v = { 3.2, 2.1, 7.6, 5.4 };
list<pair<string, string>> capitales = { {"España","Madrid"},
                                       {"Francia","París"},
                                       {"Italia","Roma"}
                                       };
```

- Para iniciar un objeto creado con **new**:

```
new vector<string>{"uno", "dos", "tres"}; // 3 elementos
```

- En una sentencia **return**:

```
return { "uno" }; // retorna una lista de un elemento
```

- Como argumento en una función:

```
fn({"uno","dos"}); // el argumento es una lista de dos elementos
```

ENUMERACIONES

Las enumeraciones tradicionales tienen el inconveniente de que sus elementos son convertidos implícitamente a **int**. Por ejemplo:

```
enum colores { rojo, verde, azul };
colores color = 2; // error: conversión de 'int' a 'colores'
int miColor = azul; // correcto: conversión de 'colores' a 'int'
```

Para solucionar este inconveniente C++0x ha añadido las enumeraciones fuertemente tipadas y delimitadas: **enum class**. Por ejemplo:

```
enum class colores { rojo, verde, azul };
int color = azul; // error: azul fuera de ámbito
int miColor = colores::azul; // error: conversión 'colores' a 'int'
```

ENTERO MUY LARGO

C++0x ha añadido el tipo **long long** para especificar un entero de al menos 64 bits. Por ejemplo:

```
long long x = 9223372036854775807LL;
```

PUNTERO NULO

Desde los comienzos de C, la constante 0 ha tenido un doble significado: constante entera y puntero constante nulo, lo cual puede dar lugar a errores. Por ejemplo, supongamos las siguientes sobrecargas de la función *fn*:

```
void fn(char *);
void fn(int);
```

Una llamada como *fn(NULL)* (donde la constante NULL está definida en C++ como 0) invocaría a *fn(int)*, que no es lo que esperamos. Para corregir esto, el estándar C++0x ha añadido la constante **nullptr** para especificar un puntero nulo. Por ejemplo:

```
fn(nullptr);
```

EXPRESIONES CONSTANTES GENERALIZADAS

Una sentencia C++ requiere en muchas ocasiones una constante. Por ejemplo, cuando se declara una matriz, la dimensión especificada tiene que ser una constante; también, en una sentencia **switch**, los **case** deben ir seguidos por una constante. Esto quiere decir que el compilador en esos casos no permitirá nada que no sea una constante. Para dar solución a situaciones como la presentada a continuación, C++0x estándar añade la palabra reservada **constexpr**. En el ejemplo siguiente, si no utilizáramos **constexpr** para especificar que la función *operator |* devuelve una constante, el **case** tercero daría un error indicando que requiere una constante.

```
enum estadoFlujo { good, fail, bad, eof };

constexpr int operator|(estadoFlujo f1, estadoFlujo f2)
{
    return estadoFlujo(f1|f2);
}

void fn(estadoFlujo x)
{
    switch (x)
    {
        case bad:
            // ...
            break;
        case eof:
            // ...
            break;
        case fail|eof: // invoca a operator(fail, eof)
            // ...
            break;
        default:
            // ...
            break;
    }
}
```

REFERENCIAS rvalue y lvalue

Un expresión *lvalue* es aquella que puede ser utilizada en el lado izquierdo de una asignación y una expresión *rvalue* es aquella que puede ser utilizada en el lado derecho de una asignación. Esto es, cuando, por ejemplo, una variable *x* o un elemento *a[i]* de una matriz se presenta como el objetivo de una operación de asignación: *x = z*, o como el operando del operador incremento: *x++*, o como el operando del operador dirección: *&x*, nosotros utilizamos el *lvalue* de la variable o

del elemento de la matriz. Esto es, el *lvalue* es la localización o dirección de memoria de esa variable o elemento de la matriz. En caso contrario, cuando la variable x o el elemento $a[i]$ de una matriz se utilizan en una expresión: $z = x + 5$, nosotros utilizamos su *rvalue*. El *rvalue* es el valor almacenado en la localización de memoria correspondiente a la variable.

Sólo las expresiones que tienen una localización en la memoria pueden tener un *lvalue*. Así, en C/C++ esta expresión no tiene sentido: $(7 + 3)++$, porque la expresión $(7 + 3)$ no tiene un *lvalue*.

En otras palabras, durante la compilación se hace corresponder *lvalues* a los identificadores y durante la ejecución, en la memoria, se hacen corresponder *rvalues* a los *lvalues*.

Una vez aclarados los conceptos *lvalue/rvalue*, vamos a estudiar las “referencias *lvalue/rvalue*”. En C++, las referencias no **const** pueden ser vinculadas a *lvalues*, pero no a *rvalues*:

```
void fn(int& a) {}
int x = 0;
fn(x);      // x es un lvalue que se vincula con la referencia 'a'
fn(0);      // error: 0 es un rvalue
```

y las referencias **const** a *lvalues* o *rvalues*:

```
void fn(const int& a) {}
int x = 0;
fn(x);      // x es un lvalue vinculado con 'a'
fn(0);      // correcto: rvalue vinculado con 'a' (const int&)
```

¿Por qué no se puede vincular un *rvalue* no **const** a una referencia no **const**? Pues para no permitir cambiar los objetos temporales que son destruidos antes de que su nuevo valor (si se pudiera cambiar) pueda ser utilizado:

```
class C {}
C fn() { C x; return x; }
// ...
C z;
C& r1 = z;          // correcto: z es un lvalue
C& r2 = fn();       // error: objeto temporal (rvalue no const)
const C& r3 = fn(); // correcto: referencia const a un rvalue
```

La función *fn* devuelve un objeto temporal, el que se utilizará en la asignación. Devolver un objeto temporal copia de un objeto fuente (en el ejemplo copia de x) en lugar de utilizar el propio objeto fuente, tiene un coste: crear el objeto

temporal y destruirlo. Para dar solución a este problema y a otros similares, el nuevo estándar C++0x introduce el concepto de *referencia rvalue*.

Una *referencia rvalue* a un objeto de la clase *C* es creada con la sintaxis *C&&*, para distinguirla de la referencia existente (*C&*). La referencia existente se denomina ahora *referencia lvalue*. La nueva referencia *rvalue* se comporta como la referencia actual *lvalue* y además puede vincularse a un *rvalue*:

```
class C {};
C fn() { C x; return x; };
// ...
C a;
C& r1 = a;           // correcto: a es un lvalue
C& r2 = fn();       // error: fn() es un rvalue (objeto temporal)
C&& rr1 = fn();     // correcto: referencia rr1 a un objeto temporal
C&& rr2 = a;        // correcto: referencia rr2 a un lvalue
```

Una referencia *rvalue* y una *lvalue* son tipos distintos. Por lo tanto, podrán ser utilizadas, una y otra, para declarar versiones sobrecargadas de la misma función. Por ejemplo:

```
class C {};

void fn1(const C& x) {}; // #1: referencia lvalue
void fn1(C&& x) {};     // #2: referencia rvalue

C fn2() { C x; return x; };
const C cfn2() { C x; return x; };

int main()
{
    C a;
    const C ca;
    fn1(a);           // llama a #1 (lvalue)
    fn1(ca);         // llama a #1 (lvalue const)
    fn1(fn2());      // llama a #2 (rvalue)
    fn1(cfn2());     // llama a #1 (rvalue const)
}
```

La primera llamada a *fn1* utiliza la referencia *lvalue* (#1) porque el argumento es un *lvalue* (la conversión *lvalue* a *rvalue* es menos afin que la de *C&* a *const C&*). La segunda llamada a *fn1* es una coincidencia exacta para #1. La tercera es una coincidencia exacta para #2. Y la cuarta, no puede utilizar #2 porque la conversión de *const C&&* a *C&&* no está permitida; llama a #1 a través de una conversión de *rvalue* a *lvalue*.

Las normas de resolución de la sobrecarga indican que los *rvalues* prefieren *referencias rvalue* (una *referencia rvalue* se vincula a *rvalues* incluso si no están calificados **const**), que los *lvalues* prefieren *referencias lvalue*, que los *rvalues* pueden vincularse a una *referencia lvalue const* (por ejemplo, *const C&*), a menos que haya una *referencia rvalue* en el conjunto de sobrecargas, y que los *lvalues* pueden vincularse a una *referencia rvalue*, pero prefieren una *referencia lvalue* si la hay.

La razón principal para añadir *referencias rvalue* es eliminar copias innecesarias de los objetos, lo que facilita la aplicación de la *semántica de mover* (no copiar: *semántica de copiar*). A diferencia de la conocida idea de copiar, mover significa que un objeto destino roba los recursos del objeto fuente, en lugar de copiarlos o compartirlos. Se preguntará, ¿y por qué iba alguien a querer eso? En la mayoría de los casos, preferiremos la semántica de copia. Sin embargo, en algunos casos, hacer una copia de un objeto es costoso e innecesario. C++ ya implementa la semántica de mover en varios lugares, por ejemplo, con **auto_ptr** y para optimizar la operación de retornar un objeto.

Para aclarar lo expuesto, piense en dos objetos **auto_ptr** *a* y *b*. Cuando realizamos la operación $b = a$, lo que sucede es que el objeto *b* pasa a ser el nuevo propietario del objeto apuntado por *a*, y *a* pasa a no apuntar a nada y es destruido.

Según lo expuesto, es eficiente añadir a una clase nuevas versiones sobrecargadas del operador de asignación y del constructor copia que utilicen la semántica de mover, ya que son más eficientes que el operador de asignación y constructor copia tradicionales. Por ejemplo:

```
class C
{
public:
    C(){ /* ... */ }
    // Semántica de copiar
    C(const C& a){ /* ... */ };
    C& operator=(const C& a){ /* ... */ };

    // Semántica de mover
    C(C&& a){ /* ... */ };
    C& operator=(C&& a){ /* ... */ };
    // ...
};
```


PLANTILLAS *variadic*

En ciencias de la computación, se dice que un operador o una función es *variadic* cuando puede tomar un número variable de argumentos. Pues bien, C++0x incluye también plantillas con un número variable de parámetros.

Por ejemplo, la siguiente plantilla P puede aceptar cero o más argumentos de tipo:

```
template<typename... T> class P
{
    P(T... t) { }
};
```

A partir de esta plantilla, $P<int>$ generará una clase más o menos así, suponiendo que el nombre de la nueva clase es P_int :

```
class P_int
{
    P_int(int t) { }
};
```

Y una expresión como $P<int, double>$ generará una clase más o menos así, suponiendo que el nombre de la nueva clase es P_int_double :

```
class P_int_double
{
    P_int_double(int t1, double t2) { }
};
```

ENVOLTORIO PARA UNA REFERENCIA

C++0x aporta la plantilla `reference_wrapper<T>` para manipular referencias a objetos. Un objeto `reference_wrapper<T>` contiene una referencia a un objeto de tipo T . Esta plantilla, entre otros, proporciona los métodos:

```
reference_wrapper<T> ref(T& t);
T& get() const;
```

Ahora, con esta plantilla, entre otras cosas, podremos trabajar con vectores de referencias en lugar de con vectores de punteros. Por ejemplo:

```
class C { /* ... */ }
```

```
int main()
{
    vector<reference_wrapper<C>> v;
    C obj1(10); C obj2(20);
    v.push_back(ref(obj1)); v.push_back(ref(obj2));
    v[1].get() = C(25); // ahora obj2.n = 25
}
```

OPERADOR `decltype`

El operador **`decltype`** evalúa el tipo de una expresión. Por ejemplo:

```
const double&& fn();
int i;
struct S { double v; };
const S* p = new S();

decltype(fn()) v1; // el tipo es const double&&
decltype(i) v2; // el tipo es int
decltype(p->v) v3; // el tipo es double
```

DECLARACIÓN DE FUNCIÓN

C++0x aporta una nueva declaración de función de la forma:

```
auto fn([parámetros])->tipo_retornado
```

El *tipo retornado* sustituirá a **`auto`**. Por ejemplo, la siguiente línea declara una función *f* que tiene un parámetro *x* de tipo **`int`** y devuelve un puntero a una matriz de 4 elementos de tipo **`double`**:

```
auto f(int x)->double(*)[4];
```

Podemos combinar este tipo de declaración con el operador **`decltype`** para deducir el tipo del valor retornado. Por ejemplo, el tipo del valor retornado por la siguiente función es `vector<T>::iterator`:

```
template <class T>
auto ultimo(vector<T>& v){ return v.end() } ->decltype(v.end());
```

PUNTEROS INTELIGENTES

Un objeto **`auto_ptr`** emplea el modelo de “propiedad exclusiva”. Esto significa que no se puede vincular más de un objeto **`auto_ptr`** a un mismo recurso. Para

asegurar la propiedad exclusiva, las operaciones de copiar y asignar del **auto_ptr** hacen que el objeto fuente entregue la propiedad de los recursos al objeto destino. Veamos un ejemplo:

```
void fn()
{
    // Crear sp1 con la propiedad del puntero a un objeto C
    auto_ptr<C> sp1(new C);
    { // Bloque interno: define sp2
        // Crear sp2 a partir sp1.
        auto_ptr<C> sp2(sp1);
        // La propiedad del puntero a C ha pasado a sp2
        // y sp1 pasa a ser nulo.
        sp2->m_fn(); // sp2 representa al objeto de la clase C
        // El destructor del sp2 elimina el objeto C apuntado
    }
    sp1->m_fn(); // error: sp1 es nulo
}
```

En el código anterior destacamos, por una parte, que después de la copia el objeto fuente ha cambiado, algo que normalmente no se espera. ¿Por qué funcionan así? Porque si no fuera así, los dos objetos **auto_ptr**, al salir de su ámbito, intentarían borrar el mismo objeto con un resultado impredecible; pero en nuestro ejemplo también ocurre que cuando *sp1* invoca a *m_fn* no puede hacerlo porque ya no tiene la propiedad del objeto *C* (es nulo); el resultado es un error de ejecución. Por otra parte, los objetos **auto_ptr** no se pueden utilizar con contenedores de la biblioteca STL porque estos pueden mover sus elementos y ya hemos visto lo que pasa cuando copiamos un objeto **auto_ptr** en otro.

Para afrontar estos problemas C++0x proporciona la plantilla **shared_ptr**. Igual que **auto_ptr**, la plantilla **shared_ptr** almacena un puntero a un objeto, pero **shared_ptr** implementa la semántica de la propiedad compartida: el último propietario del puntero es el responsable de la destrucción del objeto o de la liberación de los recursos asociados con el puntero almacenado. Un objeto **shared_ptr** está vacío si no posee un puntero. El ejemplo siguiente demuestra que los problemas que en la versión anterior introducía **auto_ptr** ahora han desaparecido:

```
void fn1()
{
    // Crear sp1 con la propiedad del puntero a un objeto C
    shared_ptr<C> sp1(new C);
    { // Bloque interno: define sp2
        // Crear sp2 a partir sp1.
        shared_ptr<C> sp2(sp1);
        // La propiedad del puntero a C es compartida entre sp2 y sp1.
        sp2->m_fn(); // sp2 representa al objeto de la clase C
        // El destructor del sp2 no elimina el objeto C apuntado
    }
```

```

    }
    spl->m_fn(); // spl representa al objeto de la clase C
    // El destructor del spl elimina el objeto C apuntado
}

```

Otra plantilla relacionada con **shared_ptr** es **weak_ptr**. Esta plantilla almacena una referencia débil a un objeto que ya está gestionado por un **shared_ptr**. Para acceder al objeto, un **weak_ptr** se puede convertir en un **shared_ptr** utilizando la función miembro **lock**.

A diferencia de **shared_ptr**, un **weak_ptr** no incrementa el contador de referencias del recurso compartido. Por ejemplo, si se tiene un **shared_ptr** y un **weak_ptr**, ambos vinculados a los mismos recursos, el contador de referencias es 1, no 2:

```

C* pObjC = new C;
shared_ptr<C> sp1(pObjC); // el contador de referencias vale 1
weak_ptr<C> wp1(sp1);    // el contador de referencias vale 1
shared_ptr<C> sp2(sp1);  // el contador de referencias vale 2
cout << "El contador de sp1/sp2 referencias es: "
     << sp1.use_count() << endl;

```

Los objetos **weak_ptr** se utilizan para romper los ciclos en las estructuras de datos. Un ciclo es similar a un abrazo mortal en el *multithreading*: dos recursos mantienen punteros entre sí de manera que un puntero no puede ser liberado porque el otro recurso comparte su propiedad y viceversa. Pues bien, esta dependencia puede ser rota utilizando **weak_ptr** en lugar de **shared_ptr**.

DELEGACIÓN DE CONSTRUCTORES

La delegación de constructores proporciona un mecanismo por el cual un constructor puede delegar en otro para realizar la iniciación de un objeto. Esto es útil cuando las distintas sobrecargas de un constructor no se pueden refundir en una que utilice parámetros con valores por omisión y estemos obligados a definir dos constructores y, probablemente, a repetir el código de iniciación.

Según lo expuesto, las dos versiones de la clase *A* mostradas a continuación serían equivalentes:

```

class D
{
public:
    operator double() const;
};

```

```

class A
{
private:
    int n;
    double d;
    void iniciar();
public:
    A(int x = 0, double y = 0.0) : n(x), d(y) { iniciar(); }
    A(D& b) : d(b) { iniciar(); }
    // ...
};

class A
{
private:
    int n;
    double d;
public:
    A(int x = 0, double y = 0.0) : n(x), d(y) { /* iniciar */ }
    A(B& b) : A(0, b) {}
    // ...
};

```

CONVERSIONES EXPLÍCITAS

Una función de conversión puede ser explícita (**explicit**), en cuyo caso se requiere la intervención directa del usuario (especificando una conversión forzada) allí donde sea necesario utilizarla. Si no es explícita, las conversiones se realizarán sin la intervención del usuario (sin tener que especificar una conversión forzada). Por ejemplo:

```

class A { };

class B
{
public:
    explicit operator A() const;
};

void fn(B b)
{
    A a1(b);    // correcto: iniciación directa
    A a2 = b;   // error: conversión B a A requerida
    A a3 = (A)b; // correcto: conversión B a A forzada
}

```

EXPRESIONES LAMBDA

Una expresión *lambda* (también conocida como función *lambda*) es una función sin nombre definida en el lugar de la llamada. Como tal, es similar a un objeto función (un objeto que puede ser invocado como si de una función ordinaria se tratara). De hecho, las expresiones *lambda* se transforman automáticamente en objetos función. Entonces, ¿por qué no utilizar los objetos función directamente? Podría hacerse así, pero la creación de un objeto función es una tarea laboriosa: hay que definir una clase con miembros de datos, una sobrecarga del operador llamada a función y un constructor. A continuación, hay que crear un objeto de ese tipo en todos los lugares donde sea requerido.

Para demostrar la utilidad de las expresiones *lambda*, supongamos que necesitamos buscar el primer objeto *X* cuyo valor se encuentra dentro de un determinado rango. Utilizando los objetos función tradicionales se puede escribir una clase *F* de objetos función así:

```
class X
{
    double d;
public:
    X(double x = 0.0) : d(x) {}
    double dato() const { return d; }
};

class F
{
    double inf, sup;
public:
    F(double i, double s) : inf(i), sup(s) {}
    bool operator()(const X& obj)
    {
        return obj.dato() >= inf && obj.dato() < sup;
    }
};

int main()
{
    vector<X> v;
    v.push_back(X(1.0)); v.push_back(X(7.0)); v.push_back(X(15.0));
    double inf = 5.0, sup = 10.0;
    vector<X>::iterator resu;
    resu = std::find_if(v.begin(), v.end(), F(inf, sup));
    cout << (*resu).dato() << endl;
}
```

Obsérvese el tercer parámetro de la función **find_if** definida en *<algorithm>*. Se trata de un objeto función que define la función a aplicar sobre objetos de otra clase. En otras palabras, la clase *F* representa la clase del objeto función que el compilador generaría para una expresión *lambda* dada. Según esto, si en su lugar utilizamos una expresión *lambda* el código quedaría así:

```
class X
{
    double d;
public:
    X(double x = 0.0) : d(x) {}
    double dato() const { return d; }
};

int main()
{
    vector<X> v;
    v.push_back(X(1.0)); v.push_back(X(7.0)); v.push_back(X(15.0));
    double inf = 5.0, sup = 10.0;
    vector<X>::iterator resu;
    resu = std::find_if(v.begin(), v.end(),
        [&](const X& obj) -> bool {
            return (obj.dato() >= inf && obj.dato() <= sup);});
    cout << (*resu).dato() << endl;
}
```

Vemos que una expresión *lambda* empieza con el presentador *lambda*, [], que puede estar vacío (no depende de variables fuera del ámbito del cuerpo de la expresión *lambda*), puede incluir el símbolo = (depende de variables que serán pasadas por valor) o el símbolo & (depende de variables que serán pasadas por referencia). A continuación, entre paréntesis, se especifican los parámetros de la expresión *lambda*. Después el tipo del valor retornado, el cual se puede omitir si no hay valor retornado o si se puede deducir de la expresión. Y finalmente, está el cuerpo de la expresión *lambda*.

A partir de una expresión *lambda* se genera una clase de objetos función. Esto es, a partir de esta expresión *lambda* se generaría una clase análoga a la clase *F* expuesta anteriormente. En resumen, la expresión *lambda* y su correspondiente clase están relacionadas así:

- Las variables con referencias externas se corresponden a los datos miembros de la clase.
- La lista de parámetros *lambda* se corresponde con la lista de los argumentos pasados a la sobrecarga del operador () de llamada a función.

- El cuerpo de la expresión *lambda* se corresponde más o menos con el cuerpo de la sobrecarga del operador ().
- El tipo del valor retornado por la sobrecarga del operador () se deduce automáticamente de una expresión **decltype**.

CONCEPTO

Mecanismo que permite especificar claramente y de manera intuitiva las limitaciones de las plantillas, mejorando al mismo tiempo la capacidad del compilador para detectar y diagnosticar violaciones de estas limitaciones.

Los *conceptos* se basan en la idea de separar la comprobación de tipos en las plantillas. Para ello, la declaración de la plantilla se incrementará con una serie de restricciones. Cuando una plantilla sea instanciada, el compilador comprobará si la instanciación reúne todas las limitaciones, o los requisitos, de la plantilla. Si todo va bien, la plantilla será instanciada; de lo contrario, un error de compilación especificará que las limitaciones han sido violadas. Veamos un ejemplo concreto. Supongamos la plantilla de función *min* definida así:

```
template<typename T>
const T& min(const T& x, const T& y)
{
    return x < y ? x : y;
}
```

Es necesario examinar el cuerpo de *min* para saber cuáles son las limitaciones de *T*. *T* debe ser un tipo que tenga, al menos, definido el operador <. Utilizando conceptos, estos requisitos pueden ser expresados directamente en la definición *min* así:

```
template<LessThanComparable T>
const T& min(const T& x, const T& y)
{
    return x < y ? x : y;
}
```

En esta otra versión de *min* vemos que en lugar de decir que *T* es un tipo arbitrario (como indica la palabra clave **typename**), se afirma que *T* es un tipo *LessThanComparable*, independientemente de lo que pueda ser. Por lo tanto, *min* sólo aceptará argumentos cuyos tipos cumplan los requisitos del concepto *LessThanComparable*, de lo contrario el compilador mostrará un error indicando que los argumentos de *min* no cumplen los requisitos *LessThanComparable*.

¿Cómo se define un concepto? Pues se define utilizando la palabra clave **concept** seguida por el nombre del concepto y de la lista de parámetros de la plantilla. Por ejemplo:

```
auto concept LessThanComparable<typename T>
{
    bool operator<(T, T);
};
```

El código anterior define un concepto llamado *LessThanComparable* que establece que el parámetro *T* de una determinada plantilla debe ser un tipo que tiene definido el operador `<`.

PROGRAMACIÓN CONCURRENTE

Una gran novedad en el estándar C++0x es el soporte para la programación concurrente. Esto es muy positivo porque ahora todos los compiladores tendrán que ajustarse al mismo modelo de memoria y proporcionar las mismas facilidades para el trabajo con hilos (*multithreading*). Esto significa que el código será portable entre compiladores y plataformas con un coste muy reducido. Esto también reducirá el número de API. El núcleo de esta nueva biblioteca es la clase **std::thread** declarada en el fichero de cabecera `<thread>`.

¿Cómo se crea un hilo de ejecución? Pues creando un objeto de la clase **thread** y vinculándolo con la función que debe ejecutar el hilo:

```
void tareaHilo();
std::thread hilo1(tareaHilo);
```

La tarea que realiza el hilo puede ser también una función con parámetros:

```
void tareaHilo(int i, std::string s, std::vector<double> v);
// ...
std::thread hilo1(tareaHilo, n, nombre, lista);
```

Los argumentos pasados se copian en el hilo antes de que la función se invoque. Ahora bien, si lo que queremos es pasarlos por referencia, entonces hay que envolverlos utilizando el método **ref**. Por ejemplo:

```
void tareaHilo(string&);
// ...
std::thread hilo1(tareaHilo, ref(s));
```

También, en lugar de definir la tarea del hilo mediante una función, la podemos definir mediante un objeto función:

```
class CTareaHilo
{
public:
    void operator()();
};

CTareaHilo tareaHilo;
std::thread hilo1(tareaHilo);
```

Evidentemente, además de la clase **thread**, disponemos de mecanismos para la sincronización de hilos: objetos de exclusión mutua (**mutex**), *locks* y variables de condición. A continuación mostramos algunos ejemplos:

```
std::mutex m;
MiClase datos;

void fn()
{
    std::lock_guard<std::mutex> bloqueo(m);
    proceso(datos);
} // El desbloqueo se produce aquí
```

Aunque los *mutex* tengan métodos para bloquear y desbloquear, en la mayoría de escenarios la mejor manera de hacerlo es utilizando *locks*. El bloqueo más simple, como vemos en el ejemplo anterior, nos lo proporciona **lock_guard**. Si lo que queremos hacer es un bloqueo diferido, o un bloqueo sin o con un tiempo de espera y desbloquear antes de que el objeto sea destruido, podemos utilizar **unique_lock**:

```
std::timed_mutex m;
MiClase datos;

void fn()
{
    std::unique_lock<std::timed_mutex>
        bloqueo(m, std::chrono::milliseconds(3)); // esperar 3 ms
    if (bloqueo) // si tenemos el bloqueo, acceder a los datos
        proceso(datos);
} // El desbloqueo se produce aquí
```

Estos es sólo una pequeña introducción a la programación con hilos. Evidentemente hay mucho más: mecanismos para esperar por eventos, almacenamiento local de hilos de ejecución, mecanismos para evitar el abrazo mortal, etc.

LA BIBLIOTECA ESTÁNDAR DE C++

La biblioteca estándar de C++ está definida en el espacio de nombres `std` y las declaraciones necesarias para su utilización son proporcionadas por un conjunto de ficheros de cabecera que se exponen a continuación. Con el fin de dar una idea general de la funcionalidad aportada por esta biblioteca, hemos clasificado estos ficheros, según su función, en los grupos siguientes:

- Entrada/Salida
- Cadenas
- Contenedores
- Iteradores
- Algoritmos
- Números
- Diagnósticos
- Utilidades generales
- Localización
- Soporte del lenguaje

La aportación que realizan los contenedores, iteradores y algoritmos a la biblioteca estándar a menudo se denomina *STL* (*Standard Template Library*, biblioteca estándar de plantillas).

A continuación mostramos un listado de los diferentes ficheros de cabecera de la biblioteca estándar, para hacernos una idea de lo que supone esta biblioteca. Un fichero de cabecera de la biblioteca estándar que comience por la letra *c* equivale a un fichero de cabecera de la biblioteca de C; esto es, un fichero `<f.h>` de la bi-

bliblioteca de C tiene su equivalente `<cf>` en la biblioteca estándar de C++ (generalmente, lo que sucede es que la implementación de `cf` incluye a `f.h`).

ENTRADA/SALIDA

<code><cstdio></code>	E/S de la biblioteca de C.
<code><cstdlib></code>	Funciones de clasificación de caracteres.
<code><wchar></code>	E/S de caracteres extendidos.
<code><fstream></code>	Flujos para trabajar con ficheros en disco.
<code><iomanip></code>	Manipuladores.
<code><ios></code>	Tipos y funciones básicos de E/S.
<code><iosfwd></code>	Declaraciones adelantadas de utilidades de E/S.
<code><iostream></code>	Objetos y operaciones sobre flujos estándar de E/S.
<code><istream></code>	Objetos y operaciones sobre flujos de entrada.
<code><ostream></code>	Objetos y operaciones sobre flujos de salida.
<code><sstream></code>	Flujos para trabajar con cadenas de caracteres.
<code><streambuf></code>	Búferes de flujos.

CADENAS

<code><cctype></code>	Examinar y convertir caracteres.
<code><cstdlib></code>	Funciones de cadena estilo C.
<code><cstring></code>	Funciones de cadena estilo C.
<code><wchar></code>	Funciones de cadena de caracteres extendidos estilo C.
<code><wctype></code>	Clasificación de caracteres extendidos.
<code><string></code>	Clases para manipular cadenas de caracteres.

CONTENEDORES

<code><bitset></code>	Matriz de bits.
<code><deque></code>	Cola de dos extremos de elementos de tipo <i>T</i> .
<code><list></code>	Lista doblemente enlazada de elementos de tipo <i>T</i> .
<code><map></code>	Matriz asociativa de elementos de tipo <i>T</i> .
<code><queue></code>	Cola de elementos de tipo <i>T</i> .
<code><set></code>	Conjunto de elementos de tipo <i>T</i> (contenedor asociativo).
<code><stack></code>	Pila de elementos de tipo <i>T</i> .
<code><vector></code>	Matriz de elementos de tipo <i>T</i> .

ITERADORES

<code><iterator></code>	Soporte para iteradores.
-------------------------------	--------------------------

ALGORITMOS

<code><algorithm></code>	Algoritmos generales (buscar, ordenar, contar, etc.).
<code><cstdlib></code>	<i>bsearch</i> y <i>qsort</i> .

NÚMEROS

<code><cmath></code>	Funciones matemáticas.
<code><complex></code>	Operaciones con números complejos.
<code><cstdlib></code>	Números aleatorios estilo C.
<code><numeric></code>	Algoritmos numéricos generalizados.
<code><valarray></code>	Operaciones con matrices numéricas.

DIAGNÓSTICOS

<code><cassert></code>	Macro ASSERT.
<code><cerrno></code>	Tratamiento de errores estilo C.
<code><exception></code>	Clase base para todas las excepciones.
<code><stdexcept></code>	Clases estándar utilizadas para manipular excepciones.

UTILIDADES GENERALES

<code><ctime></code>	Fecha y hora estilo C.
<code><functional></code>	Objetos función.
<code><memory></code>	Funciones para manipular bloques de memoria.
<code><utility></code>	Manipular pares de objetos.

LOCALIZACIÓN

<code><locale></code>	Control estilo C de las diferencias culturales.
<code><locale></code>	Control de las diferencias culturales.

SOPORTE DEL LENGUAJE

<code><float></code>	Límites numéricos en coma flotante estilo C.
<code><limits></code>	Límites numéricos estilo C.
<code><setjmp></code>	Salvar y restaurar el estado de la pila.
<code><signal></code>	Establecimiento de manejadores para condiciones excepcionales (también conocidos como señales).
<code><stdarg></code>	Lista de parámetros de función de longitud variable.
<code><stddef></code>	Soporte de la biblioteca al lenguaje C.
<code><stdlib></code>	Definición de funciones, variables y tipos comunes.
<code><time></code>	Manipulación de la fecha y hora.
<code><exception></code>	Tratamiento de excepciones.

<code><limits></code>	Límites numéricos.
<code><new></code>	Gestión de memoria dinámica.
<code><typeinfo></code>	Identificación de tipos durante la ejecución.

LA BIBLIOTECA DE C

La biblioteca de C puede ser utilizada también desde un programa C++. Por ejemplo, con frecuencia algunos programadores prefieren utilizar las funciones de E/S de C, que se encuentran en *stdio.h* (*cstdio* en la biblioteca de C++ estándar), por ser más familiares. En este caso, con la llamada a **sync_with_stdio(false)** de la clase *ios_base* antes de la primera operación de E/S puede desactivar la sincronización de las funciones *iostream* con las funciones *cstdio*, que por omisión está activada. Esta función retorna el modo de sincronización (**true** o **false**) previo.

```
bool sync_with_stdio(bool sync = true);
```

Cuando la sincronización está desactivada (*sync = false*), las operaciones de E/S con **cin**, **cout**, **cerr** y **clog** se realizan utilizando un búfer de tipo **filebuf** y las operaciones con **stdin**, **stdout** y **stderr** se realizan utilizando un búfer de tipo **stdiobuf**; esto es, los flujos *iostream* y los flujos *cstdio* operan independiente, lo cual puede mejorar la ejecución pero sin garantizar la sincronización. En cambio, cuando hay sincronización (*sync = true*), todos los flujos utilizan el mismo búfer, que es **stdiobuf**. El siguiente ejemplo le permitirá comprobar la sincronización en operaciones de E/S:

```
// Comprobar si sync_with_stdio(true) trabaja
#include <cstdio>
#include <iostream>
using namespace std;

int main()
{
    /*
    1. ¿Qué se escribe en test.txt cuando se invoca a
       sync_with_stdio con el argumento true?
    2. ¿Y con el argumento false?
    */
}
```

```
3. ¿Y cuando no se invoca a sync_with_stdio? (caso por omisión)
*/
ios_base::sync_with_stdio();

// Vincular stdout con el fichero test.txt
freopen ("test.txt", "w", stdout);

for (int i = 0; i < 2; i++)
{
    printf("1");
    cout << "2";
    putc('3', stdout);
    cout << '4';
    fputs("5", stdout);
    cout << 6;
    putchar('7');
    cout << 8 << '9';
    if (i)
        printf("0\n");
    else
        cout << "0" << endl;
}
}

/*
Resultados:
1. 1234567890
   1234567890

2. 1357246890
   13570
   24689

3. 1234567890
   1234567890
*/
```

A continuación se resumen las funciones más comunes de la biblioteca de C.

ENTRADA Y SALIDA

printf

La función **printf** escribe bytes (caracteres ASCII) de **stdout**.

```
#include <cstdio>
int printf(const char *formato[, argumento]...);
```


formato Especifica cómo va a ser la salida. Es una cadena de caracteres formada por caracteres ordinarios, secuencias de escape y especificaciones de formato. El formato se lee de izquierda a derecha.

```
unsigned int edad = 0;
float peso = 0;

// ...
printf("Tiene %u años y pesa %g kilos\n", edad, peso);
```

argumento Representa el valor o valores a escribir. Cada argumento debe tener su correspondiente especificación de formato y en el mismo orden.

```
printf("Tiene %u años y pesa %g kilos\n", edad, peso);
```

Una especificación de formato está compuesta por:

`%[flags][ancho][.precisión][{h}|l|L} tipo`

Una especificación de formato siempre comienza con %. El significado de cada uno de los elementos se indica a continuación:

flags	significado
-	Justifica el resultado a la izquierda, dentro del <i>ancho</i> especificado. Por defecto la justificación se hace a la derecha.
+	Antepone el signo + o - al valor de salida. Por defecto sólo se pone signo - a los valores negativos.
0	Rellena la salida con ceros no significativos hasta alcanzar el ancho mínimo especificado.
blanco	Antepone un espacio en blanco al valor de salida si es positivo. Si se utiliza junto con +, entonces se ignora.
#	Cuando se utiliza con la especificación de formato o , x o X , antepone al valor de salida 0 , 0x o 0X , respectivamente. Cuando se utiliza con la especificación de formato e , E o f , fuerza a que el valor de salida contenga un punto decimal en todos los casos. Cuando se utiliza con la especificación de formato g o G , fuerza a que el valor de salida contenga un punto decimal en todos los casos y evita que los ceros arrastrados sean truncados. Se ignora con c , d , i , u o s .

ancho Mínimo número de posiciones para la salida. Si el valor a escribir ocupa más posiciones de las especificadas, el ancho es incrementado en lo necesario.

precisión El significado depende del tipo de la salida.

tipo Es uno de los siguientes caracteres:

carácter **salida**

d **(int)** enteros con signo en base 10.

i **(int)** enteros con signo en base 10.

u **(int)** enteros sin signo en base 10.

o **(int)** enteros sin signo en base 8.

x **(int)** enteros sin signo en base 16 (01...abcdef).

X **(int)** enteros sin signo en base 16 (01...ABCDEF).

f **(double)** valor con signo de la forma: $[-]dddd.dddd$. El número de dígitos antes del punto decimal depende de la magnitud del número y el número de decimales de la precisión, la cual es 6 por defecto.

e **(double)** valor con signo, de la forma $[-]d.dddde[\pm]ddd$.

E **(double)** valor con signo, de la forma $[-]d.ddddE[\pm]ddd$.

g **(double)** valor con signo, en formato **f** o **e** (el que sea más compacto para el valor y precisión dados).

G **(double)** igual que **g**, excepto que **G** introduce el exponente **E** en vez de **e**.

c **(int)** un solo carácter, correspondiente al byte menos significativo.

s (*cadena de caracteres*) escribir una cadena de caracteres hasta el primer carácter nulo (`'\0'`).

Ejemplo:

```
#include <cstdio>
int main()
{
    int a = 12345;
    float b = 54.865F;
    printf("%d\n", a);           /* escribe 12345\n */
    printf("\n%10s\n%10s\n", "abc", "abcdef");
    printf("\n%-10s\n%-10s\n", "abc", "abcdef");
    printf("\n");              /* avanza a la siguiente línea */
    printf("%.2f\n", b);       /* escribe b con dos decimales */
}
```

La *precisión*, en función del tipo, tiene el siguiente significado:

d,i,u,o,x,X	Especifica el mínimo número de dígitos que se tienen que escribir. Si es necesario, se rellena con ceros a la izquierda. Si el valor excede de la precisión, no se trunca.
e,E,f	Especifica el número de dígitos que se tienen que escribir después del punto decimal. Por defecto es 6. El valor es redondeado.
g,G	Especifica el máximo número de dígitos significativos (por defecto 6) que se tienen que escribir.
c	La precisión no tiene efecto.
s	Especifica el máximo número de caracteres que se escribirán. Los caracteres que excedan este número, se ignoran.

h	Se utiliza como prefijo con los tipos d , i , o , x y X , para especificar que el argumento es short int , o con u para especificar un short unsigned int .
l	Se utiliza como prefijo con los tipos d , i , o , x y X , para especificar que el argumento es long int , o con u para especificar un long unsigned int . También se utiliza con los tipos e , E , f , g y G para especificar un double antes que un float .
L	Se utiliza como prefijo con los tipos e , E , f , g y G , para especificar long double . Este prefijo no es compatible con ANSI C.

scanf

La función **scanf** lee bytes (caracteres ASCII) de **stdin**.

```
#include <cstdio>
int scanf(const char *formato[, argumento]...);
```

formato Interpreta cada dato de entrada. Está formado por caracteres que genéricamente se denominan espacios en blanco (' ', \t, \n), caracteres ordinarios y especificaciones de formato. El formato se lee de izquierda a derecha.

Cada argumento debe tener su correspondiente especificación de formato y en el mismo orden (vea también la función **printf**).

Si un carácter en **stdin** no es compatible con el tipo especificado por el formato, la entrada de datos se interrumpe.

argumento Es la variable pasada por referencia que se quiere leer.

Cuando se especifica más de un argumento, los valores tecleados en la entrada hay que separarlos por uno o más espacios en blanco (' ', \t, \n), o por el carácter que se especifique en el formato. Por ejemplo:

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

Entrada de datos:

```
5 23.4 z [Entrar]
```

o también:

```
5 [Entrar]
23.4 [Entrar]
z [Entrar]
```

Una especificación de formato está compuesta por:

```
%[*][ancho][{h/l}]tipo
```

Una especificación de formato siempre comienza con `%`. El resto de los elementos que puede incluir se explican a continuación:

***** Un *asterisco* a continuación del símbolo `%` suprime la asignación del siguiente dato en la entrada.

ancho Máximo número de caracteres a leer de la entrada. Los caracteres en exceso no se tienen en cuenta.

h Se utiliza como prefijo con los tipos **d**, **i**, **n**, **o** y **x** para especificar que el argumento es **short int**, o con **u** para especificar que es **short unsigned int**.

l Se utiliza como prefijo con los tipos **d**, **i**, **n**, **o** y **x** para especificar que el argumento es **long int**, o con **u** para especificar que es **long unsigned int**. También se utiliza con los tipos **e**, **f** y **g** para especificar que el argumento es **double**.

tipo El tipo determina cómo tiene que ser interpretado el dato de entrada: como un carácter, como una cadena de caracteres o como un número. El formato más simple contiene el símbolo `%` y el *tipo*. Por ejemplo, `%i`. Los tipos que se pueden utilizar son los siguientes:

El argumento es		
Carácter	un puntero a	Entrada esperada
d	int	enteros con signo en base 10.
o	int	enteros con signo en base 8.
x, X	int	enteros con signo en base 16.

i	int	enteros con signo en base 10, 16 u 8. Si el entero comienza con 0 , se toma el valor en octal y si empieza con 0x o 0X , el valor se toma en hexadecimal.
u	unsigned int	enteros sin signo en base 10.
f		
e, E		
g, G	float	valor con signo de la forma $[-]d.ddd[\{e E\} [\pm]ddd]$
c	char	un solo carácter.
s	char	cadena de caracteres.

getchar

Leer un carácter de la entrada estándar (**stdin**).

```
#include <stdio>
int getchar(void);

char car;
car = getchar();
```

putchar

Escribir un carácter en la salida estándar (**stdout**).

```
#include <stdio>
int putchar(int c);

putchar('\n');
putchar(car);
```

gets

Leer una cadena de caracteres de **stdin**.

```
#include <stdio>
char *gets(char *var);

char nombre[41];
gets(nombre);
printf("%s\n", nombre);
```

puts

Escribir una cadena de caracteres en **stdout**.

```
#include <cstdio>
int puts(const char *var);

char nombre[41];
gets(nombre);
puts(nombre);
```

CADENAS DE CARACTERES

strcat

Añade la *cadena2* a la *cadena1*. Devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strcat(char *cadena1, const char *cadena2);
```

strcpy

Copia la *cadena2*, incluyendo el carácter de terminación nulo, en la *cadena1*. Devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strcpy(char *cadena1, const char *cadena2);
```

```
char cadena[81];
strcpy(cadena, "Hola. ");
strcat(cadena, "Hasta luego.");
```

strchr

Devuelve un puntero a la primera ocurrencia de *c* en *cadena* o un valor **NULL** si el carácter no es encontrado. El carácter *c* puede ser el carácter nulo ('\0').

```
#include <cstring>
char *strchr(const char *cadena, int c);
```

```
char *pdest;
pdest = strchr(cadena, car);
```

strrchr

Devuelve un puntero a la última ocurrencia de *c* en *cadena* o un valor **NULL** si el carácter no se encuentra. El carácter *c* puede ser un carácter nulo ('\0').

```
#include <cstring>
char *strrchr(const char *cadena, int c);
```

strcmp

Compara la *cadena1* con la *cadena2* lexicográficamente y devuelve un valor:

<0 si la *cadena1* es menor que la *cadena2*,
=0 si la *cadena1* es igual a la *cadena2* y
>0 si la *cadena1* es mayor que la *cadena2*.

Diferencia las letras mayúsculas de las minúsculas.

```
#include <cstring>
int strcmp(const char *cadena1, const char *cadena2);

resu = strcmp(cadena1, cadena2);
```

strcspn

Da como resultado la posición (subíndice) del primer carácter de *cadena1*, que pertenece al conjunto de caracteres contenidos en *cadena2*.

```
#include <cstring>
size_t strcspn(const char *cadena1, const char *cadena2);

pos = strcspn(cadena, "abc");
```

strlen

Devuelve la longitud en bytes de *cadena*, no incluyendo el carácter de terminación nulo. El tipo **size_t** es sinónimo de **unsigned int**.

```
#include <cstring>
size_t strlen(char *cadena);

char cadena[80] = "Hola";
printf("El tamaño de cadena es %d\n", strlen(cadena));
```

strncat

Añade los primeros *n* caracteres de *cadena2* a la *cadena1*, termina la cadena resultante con el carácter nulo y devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strncat(char *cadena1, const char *cadena2, size_t n);
```

strncpy

Copia n caracteres de la *cadena2* en la *cadena1* (sobrescribiendo los caracteres de *cadena1*) y devuelve un puntero a *cadena1*.

```
#include <cstring>
char *strncpy(char *cadena1, const char *cadena2, size_t n);
```

strncmp

Compara lexicográficamente los primeros n caracteres de *cadena1* y de *cadena2*, distinguiendo mayúsculas y minúsculas, y devuelve un valor:

<0 si la *cadena1* es menor que la *cadena2*,
=0 si la *cadena1* es igual a la *cadena2* y
>0 si la *cadena1* es mayor que la *cadena2*.

```
#include <cstring>
int strncmp(const char *cadena1, const char *cadena2, size_t n);
```

strspn

Da como resultado la posición (subíndice) del primer carácter de *cadena1*, que no pertenece al conjunto de caracteres contenidos en *cadena2*.

```
#include <cstring>
size_t strspn(const char *cadena1, const char *cadena2);
```

strstr

Devuelve un puntero a la primera ocurrencia de *cadena2* en *cadena1* o un valor **NULL** si la *cadena2* no se encuentra en la *cadena1*.

```
#include <cstring>
char *strstr(const char *cadena1, const char *cadena2);
```

strtok

Permite obtener de la *cadena1* los elementos en los que se divide según los delimitadores especificados en *cadena2*.

Para obtener el primer elemento, **strtok** debe tener *cadena1* como primer argumento y para obtener los siguientes elementos, debe tener **NULL**. Cada llama-

da a **strtok** devuelve un puntero al siguiente elemento o **NULL** si no hay más elementos.

```
#include <cstring>
char *strtok(char *cadena1, const char *cadena2);

#include <cstdio>
#include <cstring>

int main(void)
{
    char cadena[] = "Esta cadena, está formada por varias palabras";
    char *elemento;
    elemento = strtok(cadena, " ,");
    while (elemento != NULL)
    {
        printf("%s\n", elemento);
        elemento = strtok(NULL, " ,");
    }
}
```

strlwr

Convierte las letras mayúsculas de *cadena* en minúsculas. El resultado es la propia cadena en minúsculas.

```
#include <cstring>
char *strlwr(char *cadena);
```

strupr

Convierte las letras minúsculas de *cadena* en mayúsculas. El resultado es la propia cadena en mayúsculas.

```
#include <cstring>
char *strupr(char *cadena);
```

CONVERSIÓN DE DATOS

atof

Convierte una cadena de caracteres a un valor de tipo **double**.

```
#include <cstdlib>
double atof(const char *cadena);
```

atoi

Convierte una cadena de caracteres a un valor de tipo **int**.

```
#include <cstdlib>
int atoi(const char *cadena);
```

atol

Convierte una cadena de caracteres a un valor de tipo **long**.

```
#include <cstdlib>
long atol(const char *cadena);
```

Cuando las funciones **atof**, **atoi** y **atol** toman de la variable *cadena* un carácter que no es reconocido como parte de un número, interrumpen la conversión.

sprintf

Convierte los valores de los argumentos especificados a una cadena de caracteres que almacena en *buffer*. Devuelve como resultado un entero correspondiente al número de caracteres almacenados en *buffer* sin contar el carácter nulo de terminación.

```
#include <cstdio>
int sprintf(char *buffer, const char *formato [, argumento] ...);

#include <cstdio>
int main(void)
{
    char buffer[200], s[] = "ordenador", c = '/';
    int i = 40, j;
    float f = 1.414214F;

    j = sprintf(buffer, "\tCadena: %s\n", s);
    j += sprintf(buffer + j, "\tCarácter: %c\n", c);
    j += sprintf(buffer + j, "\tEntero: %d\n", i);
    j += sprintf(buffer + j, "\tReal: %f\n", f);
    printf("Salida:\n%s\nNúmero de caracteres = %d\n", buffer, j);
}
```

toascii

Pone a 0 todos los bits de *c*, excepto los siete bits de menor orden. Dicho de otra forma, convierte *c* a un carácter ASCII.

```
#include <cctype>
int toascii(int c);
```

tolower

Convierte c a una letra minúscula, si procede.

```
#include <cstdlib>
int tolower(int c);
```

toupper

Convierte c a una letra mayúscula, si procede.

```
#include <cstdlib>
int toupper(int c);
```

FUNCIONES MATEMÁTICAS

acos

Da como resultado el arco, en el rango 0 a π , cuyo coseno es x . El valor de x debe estar entre -1 y 1 ; de lo contrario se obtiene un error (argumento fuera del dominio de la función).

```
#include <cmath>
double acos(double x);
```

asin

Da como resultado el arco, en el rango $-\pi/2$ a $\pi/2$, cuyo seno es x . El valor de x debe estar entre -1 y 1 ; si no, se obtiene un error (argumento fuera del dominio de la función).

```
#include <cmath>
double asin(double x);
```

atan

Da como resultado el arco, en el rango $-\pi/2$ a $\pi/2$, cuya tangente es x .

```
#include <cmath>
double atan(double x);
```

atan2

Da como resultado el arco, en el rango $-\pi$ a π , cuya tangente es y/x . Si ambos argumentos son 0, se obtiene un error (argumento fuera del dominio de la función).

```
#include <cmath>
double atan2(double y, double x);
```

COS

Da como resultado el coseno de x (x en radianes).

```
#include <cmath>
double cos(double x);
```

sin

Da como resultado el seno de x (x en radianes).

```
#include <cmath>
double sin(double x);
```

tan

Da como resultado la tangente de x (x en radianes).

```
#include <cmath>
double tan(double x);
```

cosh

Da como resultado el coseno hiperbólico de x (x en radianes).

```
#include <cmath>
double cosh(double x);
```

sinh

Da como resultado el seno hiperbólico de x (x en radianes).

```
#include <cmath>
double sinh(double x);
```

tanh

Da como resultado la tangente hiperbólica de x (x en radianes).

```
#include <cmath>
double tanh(double x);
```

exp

Da como resultado el valor de e^x ($e = 2.718282$).

```
#include <cmath>
double exp(double x);
```

log

Da como resultado el logaritmo natural de x .

```
#include <cmath>
double log(double x);
```

log10

Da como resultado el logaritmo en base 10 de x .

```
#include <cmath>
double log10(double x);
```

ceil

Da como resultado un valor **double**, que representa el entero más pequeño que es mayor o igual que x .

```
#include <cmath>
double ceil(double x);
```

```
double x = 2.8, y = -2.8;
printf("%g %g\n", ceil(x), ceil(y)); // resultado: 3 -2
```

fabs

Da como resultado el valor absoluto de x . El argumento x es un valor real en doble precisión. Igualmente, **abs** y **labs** dan el valor absoluto de un **int** y un **long**, respectivamente.

```
#include <cmath>
double fabs(double x);
```

floor

Da como resultado un valor **double**, que representa el entero más grande que es menor o igual que x .

```
#include <cmath>
double floor(double x);

double x = 2.8, y = -2.8;
printf("%g %g\n", floor(x), floor(y)); // resultado: 2 -3
```

pow

Da como resultado x^y . Si x es 0 e y negativo o si x e y son 0 o si x es negativo e y no es entero, se obtiene un error (argumento fuera del dominio de la función). Si x^y da un resultado superior al valor límite para el tipo **double**, el resultado es este valor límite (1.79769e+308).

```
#include <cmath>
double pow(double x, double y);

double x = 2.8, y = -2.8;
printf("%g\n", pow(x, y)); // resultado: 0.0559703
```

sqrt

Da como resultado la raíz cuadrada de x . Si x es negativo, ocurre un error (argumento fuera del dominio de la función).

```
#include <cmath>
double sqrt(double x);
```

rand

Da como resultado un número pseudoaleatorio entero, entre 0 y **RAND_MAX** (32767).

```
#include <cstdlib>
int rand(void);
```

srand

Fija el punto de comienzo para generar números pseudoaleatorios; en otras palabras, inicia el generador de números pseudoaleatorios en función del valor de su argumento. Cuando esta función no se utiliza, el valor del primer número pseudoaleatorio generado siempre es el mismo para cada ejecución (corresponde a un argumento de valor 1).

```
#include <cstdlib>
void srand(unsigned int arg);
```

FUNCIONES DE FECHA Y HORA

clock

Indica el tiempo empleado por el procesador en el proceso en curso.

```
#include <ctime>
clock_t clock(void);
```

El tiempo expresado en segundos se obtiene al dividir el valor devuelto por **clock** entre la constante *CLOCKS_PER_SEC*. Si no es posible obtener este tiempo, la función **clock** devuelve el valor (**clock_t**)-1. El tipo **clock_t** está declarado así:

```
typedef long clock_t;
```

time

Retorna el número de segundos transcurridos desde las 0 horas del 1 de enero de 1970.

```
#include <ctime>
time_t time(time_t *seg);
```

El tipo **time_t** está definido así:

```
typedef long time_t;
```

El argumento puede ser **NULL**. Según esto, las dos sentencias siguientes para obtener los segundos transcurridos son equivalentes:

```
time_t segundos;
time(&segundos);
segundos = time(NULL);
```

ctime

Convierte un tiempo almacenado como un valor de tipo **time_t**, en una cadena de caracteres de la forma:

```
Thu Jul 08 12:01:29 2010\n\n0
```

```
#include <ctime>
char *ctime(const time_t *seg);
```

Esta función devuelve un puntero a la cadena de caracteres resultante o un puntero nulo si *seg* representa un dato anterior al 1 de enero de 1970. Por ejemplo, el siguiente programa presenta la fecha actual y, a continuación, genera cinco números pseudoaleatorios, uno cada segundo.

```
/****** Generar un número aleatorio cada segundo *****/
#include <cstdio>
#include <cstdlib>
#include <ctime>
int main()
{
    long x, tm;
    time_t segundos;
    time(&segundos);
    printf("\n%s\n", ctime(&segundos));
    srand((unsigned)time(NULL));

    for (x = 1; x <= 5; x++)
    {
        do // tiempo de espera igual a 1 segundo
            tm = clock();
        while (tm/CLOCKS_PER_SEC < x);
        // Se genera un número aleatorio cada segundo
        printf("Iteración %ld: %d\n", x, rand());
    }
}
```

localtime

Convierte el número de segundos transcurridos desde las 0 horas del 1 de enero de 1970, valor obtenido por la función **time**, a la fecha y hora correspondiente (corregida en función de la zona horaria en la que nos encontremos). El resultado es almacenado en una estructura de tipo **tm**, definida en *ctime*.

```
#include <ctime>
struct tm *localtime(const time_t *seg);
```


La función **localtime** devuelve un puntero a la estructura que contiene el resultado o un puntero nulo si el tiempo no puede ser interpretado. Los miembros de la estructura son los siguientes:

Campo	Valor almacenado
tm_sec	Segundos (0 - 59).
tm_min	Minutos (0 - 59).
tm_hour	Horas (0 - 23).
tm_mday	Día del mes (1 - 31).
tm_mon	Mes (0 - 11; enero = 0).
tm_year	Año (actual menos 1900).
tm_wday	Día de la semana (0 - 6; domingo = 0).
tm_yday	Día del año (0 - 365; 1 de enero = 0).

El siguiente ejemplo muestra cómo se utiliza esta función.

```
#include <cstdio>
#include <ctime>
int main()
{
    struct tm *fh;
    time_t segundos;

    time(&segundos);
    fh = localtime(&segundos);
    printf("%d horas, %d minutos\n", fh->tm_hour, fh->tm_min);
}
```

La función **localtime** utiliza una variable de tipo **static struct tm** para realizar la conversión y lo que devuelve es la dirección de esa variable.

MANIPULAR BLOQUES DE MEMORIA

memset

Permite iniciar un bloque de memoria.

```
#include <cstring>
void *memset(void *destino, int b, size_t nbytes);
```

El argumento *destino* es la dirección del bloque de memoria que se desea iniciar, *b* es el valor empleado para iniciar cada byte del bloque y *nbytes* es el número de bytes del bloque que se iniciarán. Por ejemplo, el siguiente código inicia a 0 la matriz *a*:

```
double a[10][10];
// ...
memset(a, 0, sizeof(a));
```

memcpy

Copia un bloque de memoria en otro.

```
#include <cstring>
void *memcpy(void *destino, const void *origen, size_t nbytes);
```

El argumento *destino* es la dirección del bloque de memoria destino de los datos, *origen* es la dirección del bloque de memoria origen de los datos y *nbytes* es el número de bytes que se copiarán desde el origen al destino. Por ejemplo, el siguiente código copia la matriz *a* en *b*:

```
double a[10][10], b[10][10];
// ...
memcpy(b, a, sizeof(a));
```

memcmp

Compara byte a byte dos bloques de memoria.

```
#include <cstring>
int memcmp(void *bm1, const void *bm2, size_t nbytes);
```

Los argumentos *bm1* y *bm2* son las direcciones de los bloques de memoria a comparar y *nbytes* es el número de bytes que se compararán. El resultado devuelto por la función es el mismo que se expuso para **strcmp**. Por ejemplo, el siguiente código compara la matriz *a* con la *b*:

```
double a[10][10], b[10][10];
// ...
if (memcmp(a, b, sizeof(a)) == 0)
    printf("Las matrices a y b contienen los mismos datos\n");
else
    printf("Las matrices a y b no contienen los mismos datos\n");
```

ASIGNACIÓN DINÁMICA DE MEMORIA

malloc

Permite asignar un bloque de memoria de *nbytes* consecutivos en memoria para almacenar uno o más objetos de un tipo cualquiera. Esta función devuelve un puntero genérico (**void ***) que referencia el espacio asignado. Si no hay suficiente es-

pacio de memoria, la función **malloc** retorna un puntero nulo (valor **NULL**) y si el argumento *nbytes* es 0, asigna un bloque de tamaño 0 devolviendo un puntero válido.

```
#include <cstdlib>
void *malloc(size_t nbytes);
```

free

Permite liberar un bloque de memoria asignado por las funciones **malloc**, **calloc** o **realloc** (estas dos últimas las veremos a continuación), pero no pone el puntero a **NULL**. Si el puntero que referencia el bloque de memoria que deseamos liberar es nulo, la función **free** no hace nada.

```
#include <cstdlib>
void free(void *vpuntero);
```

realloc

Permite cambiar el tamaño de un bloque de memoria previamente asignado.

```
#include <cstdlib>
void *realloc(void *pBlomem, size_t nBytes);
```

<i>pBlomem</i>	<i>nBytes</i>	<i>Acción</i>
NULL	0	Asigna 0 bytes (igual que malloc).
NULL	Distinto de 0	Asigna <i>nBytes</i> bytes (igual que malloc). Si no es posible, devuelve NULL .
Distinto de NULL	0	Devuelve NULL y libera el bloque original.
Distinto de NULL	Distinto de 0	Reasigna <i>nBytes</i> bytes. El contenido del espacio conservado no cambia. Si la reasignación no es posible, devuelve NULL y el bloque original no cambia.

FICHEROS

fopen

Permite crear un flujo desde un fichero, hacia un fichero o bien desde y hacia un fichero. En términos más simplificados, permite abrir un fichero para leer, para escribir o para leer y escribir.

```
#include <stdio>
FILE *fopen(const char *nomfi, const char *modo);
```

nomfi es el nombre del fichero y *modo* especifica cómo se va a abrir el fichero:

Modo	Descripción
"r"	Abrir un fichero para leer. Si el fichero no existe o no se encuentra, se obtiene un error.
"w"	Abrir un fichero para escribir. Si el fichero no existe, se crea; y si existe, su contenido se destruye para ser creado de nuevo.
"a"	Abrir un fichero para añadir información al final del mismo. Si el fichero no existe, se crea.
"r+"	Abrir un fichero para leer y escribir. El fichero debe existir.
"w+"	Abrir un fichero para escribir y leer. Si el fichero no existe, se crea; y si existe, su contenido se destruye para ser creado de nuevo.
"a+"	Abrir un fichero para leer y añadir. Si el fichero no existe, se crea.

```
FILE *pf;
pf = fopen("datos", "w"); // abrir el fichero datos
```

freopen

Desvincula el dispositivo o fichero actualmente asociado con el flujo referenciado por *pflujo* y reasigna *pflujo* al fichero identificado por *nomfi*.

```
#include <cstdio>
FILE *freopen(const char *nomfi, const char *modo, FILE *pflujo);

pf = freopen("datos", "w", stdout);
```

fclose

Cierra el flujo referenciado por *pf*.

```
#include <cstdio>
int fclose(FILE *pf);
```

ferror

Verifica si ocurrió un error en la última operación de E/S.

```
#include <cstdio>
int ferror(FILE *pf);
```

clearerr

Pone a 0 los bits de error que estén a 1, incluido el bit de fin de fichero.

```
#include <cstdio>
void clearerr(FILE *pf);

if (ferror(pf))
{
    printf("Error al escribir en el fichero\n");
    clearerr(pf);
}
```

feof

Devuelve un valor distinto de 0 cuando se intenta leer más allá de la marca *eof* (*end of file* - fin de fichero), no cuando se lee el último registro. En otro caso devuelve un 0.

```
#include <cstdio>
int feof(FILE *pf);

while (!feof(pf)) // mientras no se llegue al final del fichero
{
    // Leer aquí el siguiente registro del fichero
}
fclose(pf);
```

ftell

Devuelve la posición actual en el fichero asociado con *pf* del puntero de L/E, o bien el valor $-1L$ si ocurre un error. Esta posición es relativa al principio del fichero.

```
#include <cstdio>
long ftell(FILE *pf);
```

fseek

Mueve el puntero de L/E del fichero asociado con *pf* a una nueva localización desplazada *desp* bytes (un valor positivo avanza el puntero y un valor negativo lo retrocede) de la posición especificada por el argumento *pos*, la cual puede ser una de las siguientes:

SEEK_SET Hace referencia a la primera posición en el fichero.

SEEK_CUR Hace referencia a la posición actual del puntero de L/E.
SEEK_END Hace referencia a la última posición en el fichero.

```
#include <cstdio>
int fseek(FILE *pf, long desp, int pos);

// Calcular el nº total de registros un fichero
fseek(pf, 0L, SEEK_END);
totalreg = (int)ftell(pf)/sizeof(registro);
```

rewind

Mueve el puntero de L/E al principio del fichero asociado con *pf*.

```
#include <cstdio>
void rewind(FILE *pf);
```

fputc

Escribe un carácter *car* en la posición indicada por el puntero de lectura/escritura (L/E) del fichero o dispositivo asociado con *pf*.

```
#include <cstdio>
int fputc(int car, FILE *pf);
```

fgetc

Lee un carácter de la posición indicada por el puntero de L/E del fichero o dispositivo asociado con *pf* y avanza al siguiente carácter a leer. Devuelve el carácter leído o un **EOF**, si ocurre un error o se detecta el final del fichero.

```
#include <cstdio>
int fgetc(FILE *pf);
```

fputs

Permite copiar una cadena de caracteres en un fichero o dispositivo.

```
#include <cstdio>
int fputs(const char *cadena, FILE *pf);
```

fgets

Permite leer una cadena de caracteres de un fichero o dispositivo. Devuelve **NULL** si ocurre un error.

```
#include <stdio>
char *fgets(char *cadena, int n, FILE *pf);
```

fprintf

Permite escribir sus argumentos, con el formato especificado, en un fichero o dispositivo.

```
#include <stdio>
int fprintf(FILE *pf, const char *formato[, arg]...);
```

fscanf

Permite leer los argumentos especificados, con el formato especificado, desde un fichero o dispositivo. Devuelve un **EOF** si se detecta el final del fichero.

```
#include <stdio>
int fscanf(FILE *pf, const char *formato[, arg]...);
```

fwrite

Permite escribir *c* elementos de longitud *n* bytes almacenados en el *buffer* especificado, en el fichero asociado con *pf*.

```
#include <stdio>
size_t fwrite(const void *buffer, size_t n, size_t c, FILE *pf);
```

```
FILE *pf1 = NULL, *pf2 = NULL;
char car, cadena[36];
gets(cadena); car = getchar();
// ...
fwrite(&car, sizeof(char), 1, pf1);
fwrite(cadena, sizeof(cadena), 1, pf2);
```

fread

Permite leer *c* elementos de longitud *n* bytes del fichero asociado con *pf* y los almacena en el *buffer* especificado.

```
#include <stdio>
```

```
size_t fread(void *buffer, size_t n, size_t c, FILE *pf);

FILE *pf1 = NULL, *pf2 = NULL;
char car, cadena[36];
// ...
fread(&car, sizeof(char), 1, pf);
fread(cadena, sizeof(cadena), 1, pf);
```

fflush

Escribe en el fichero asociado con el flujo apuntado por *pf* el contenido del *buffer* definido para este flujo. En Windows, no en UNIX, si el fichero en lugar de estar abierto para escribir está abierto para leer, **fflush** borra el contenido del *buffer*.

```
#include <cstdio>
int fflush(FILE *pf);
```

MISCELÁNEA

system

Envía una orden al sistema operativo.

```
#include <cstdlib>
int system(const char *cadena-de-caracteres);

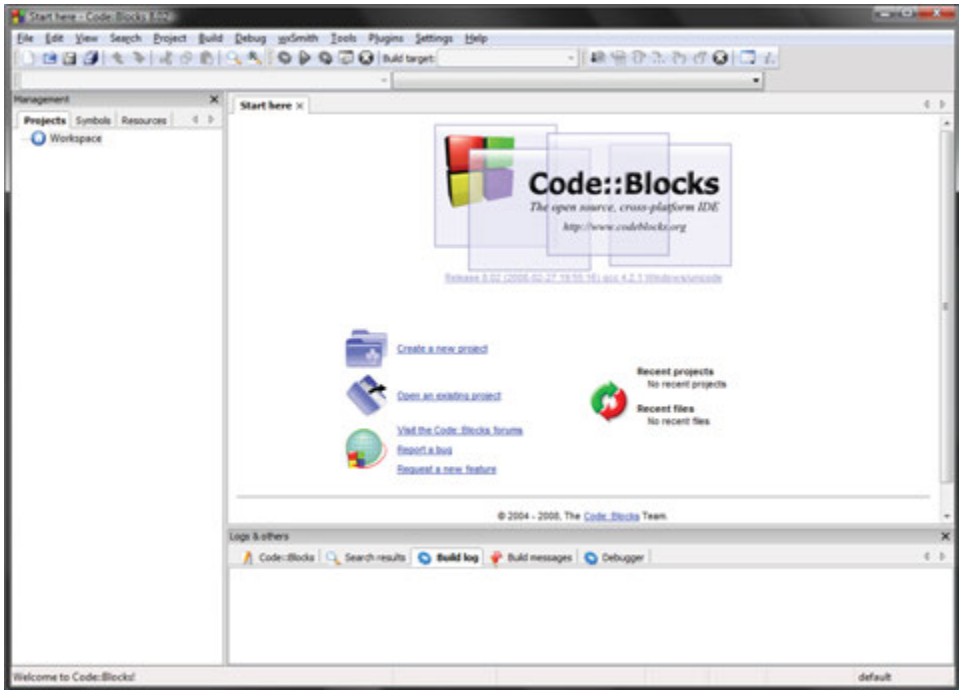
system("cls"); // limpiar la pantalla en Windows
system("clear"); // limpiar la pantalla en UNIX
```


ENTORNOS DE DESARROLLO

Cuando se utiliza un entorno de desarrollo integrado (EDI), lo primero que hay que hacer una vez instalado es asegurarse de que las rutas donde se localizan las herramientas, las bibliotecas, la documentación y los ficheros fuente hayan sido establecidas; algunos EDI sólo requieren la ruta donde se instaló el compilador. Este proceso normalmente se ejecuta automáticamente durante el proceso de instalación de dicho entorno. Si no es así, el entorno proporcionará algún menú con las órdenes apropiadas para realizar dicho proceso. Por ejemplo, en el EDI que se presenta a continuación puede comprobar esto a través de las opciones del menú *Settings*.

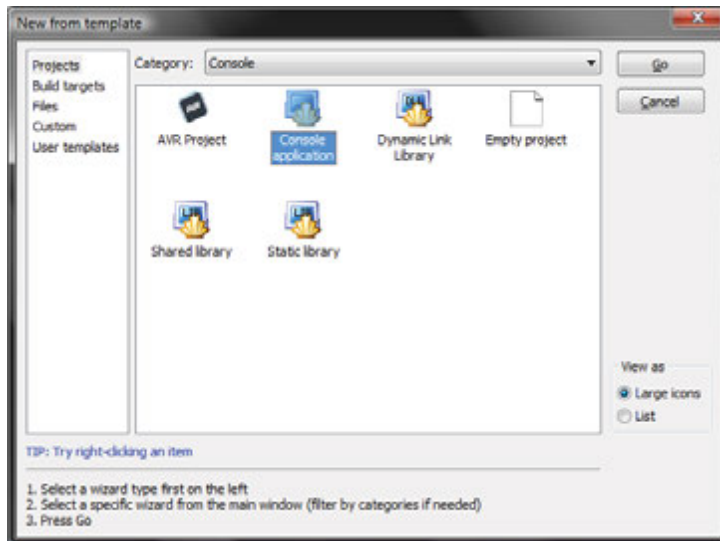
CodeBlocks

En la figura siguiente se puede observar el aspecto del entorno de desarrollo integrado *CodeBlocks* incluido en el CD que acompaña al libro.

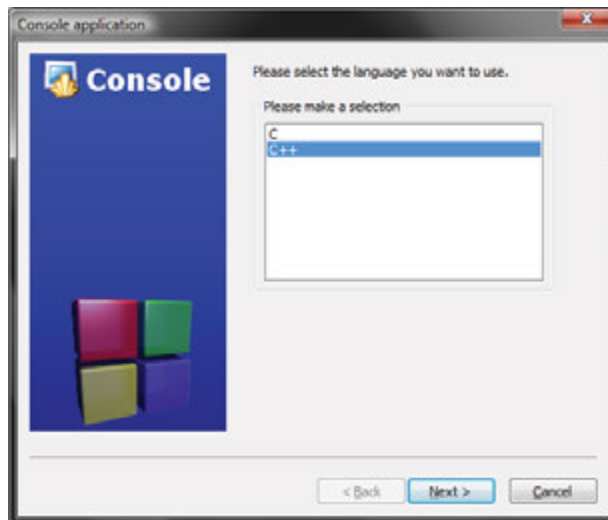


Para editar y ejecutar el programa *HolaMundo.cpp* visto en el capítulo 1, o cualquier otro programa, utilizando este entorno de desarrollo integrado, los pasos a seguir se indican a continuación:

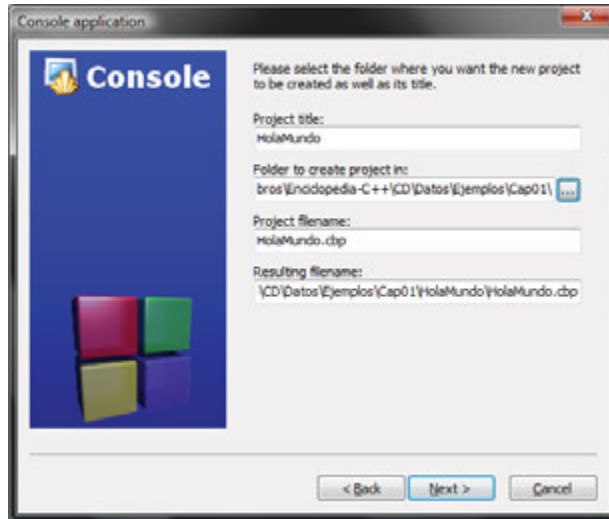
1. Suponiendo que ya está visualizado el entorno de desarrollo, creamos un nuevo proyecto C++ (*File, New, Project*). Se muestra la ventana siguiente:



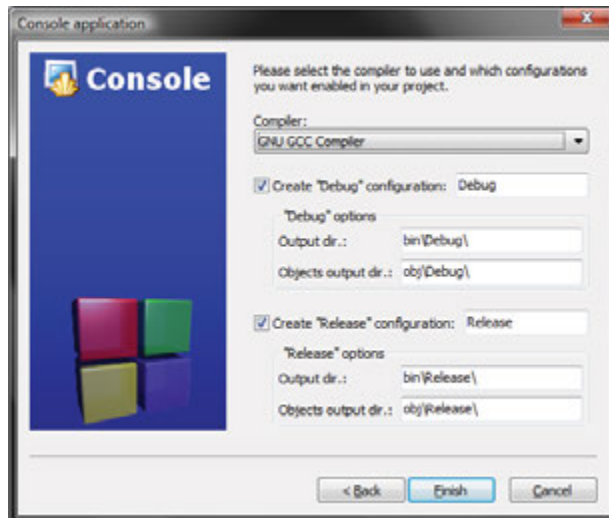
2. Elegimos la categoría consola (*Console*), la plantilla *Console application* y pulsamos el botón *Go*.



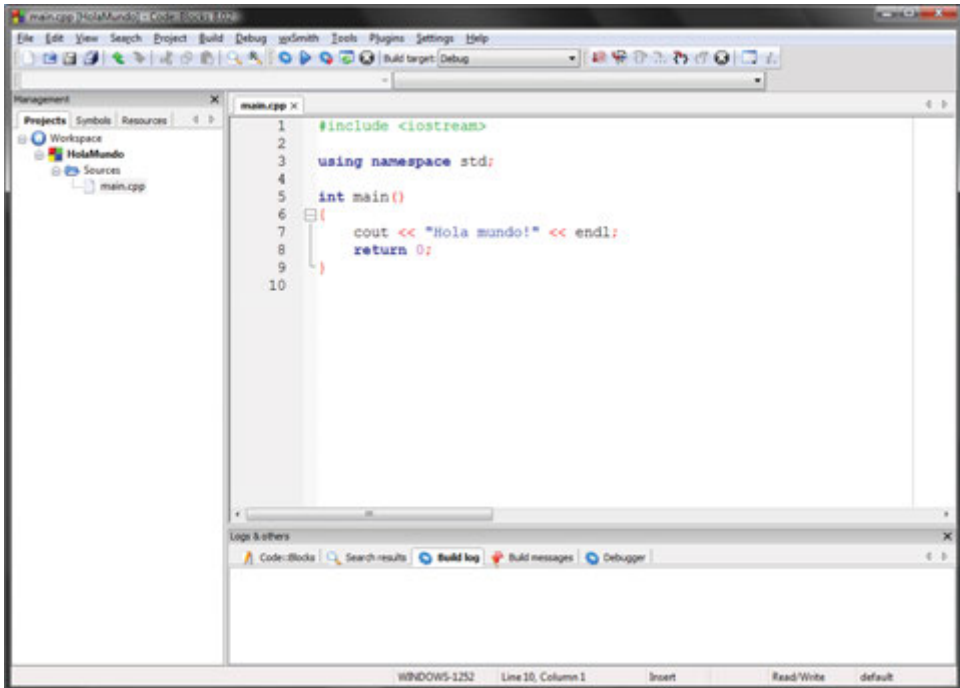
3. Seleccionamos el lenguaje C++ y hacemos clic en el botón *Next*.



4. Especificamos el nombre del proyecto, la carpeta donde será guardado y hacemos clic en *Next*.



5. Si los datos presentados en la ventana anterior son correctos, hacemos clic en el botón *Finish*. El proyecto está creado; contiene un fichero *main.cpp* que incluye la función **main** por donde se iniciará y finalizará la ejecución del programa.

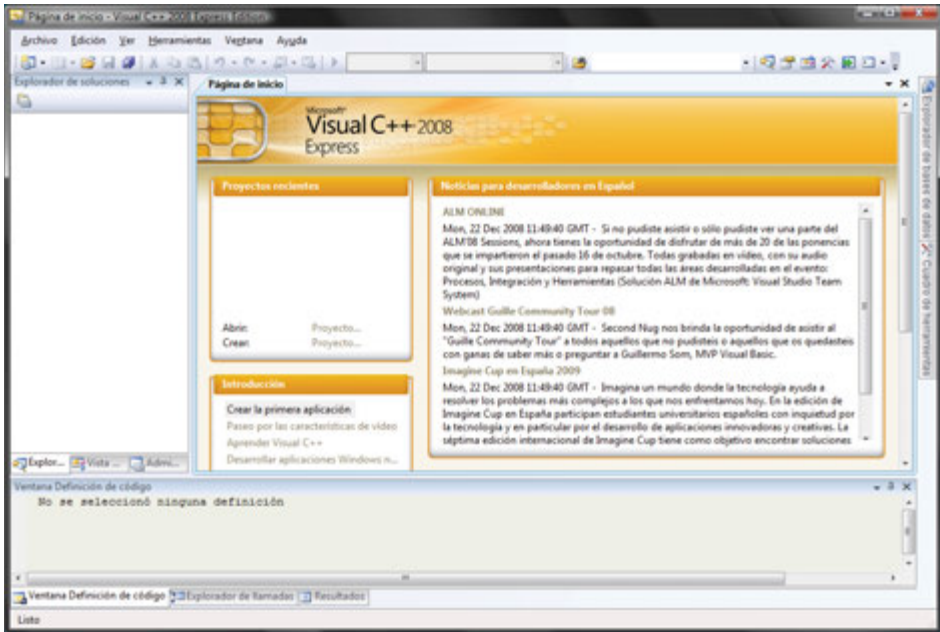


6. A continuación, según se puede observar en la figura anterior, editamos el código que compone el programa y lo guardamos.
7. Después, para compilar el programa, ejecutamos la orden *Build* del menú *Build* y, una vez compilado (sin errores), lo podemos ejecutar seleccionando la orden *Run* del mismo menú (si no pudiéramos ver la ventana con los resultados porque desaparece -no es el caso-, añadiríamos al final de la función **main**, antes de **return**, la sentencia “`system("pause");`” y al principio del fichero *.cpp* la directriz `#include <cstdlib>`, si fuera necesario).

En el caso de que la aplicación esté compuesta por varios ficheros fuente, simplemente tendremos que añadirlos al proyecto ejecutando la orden *File* del menú *File*.

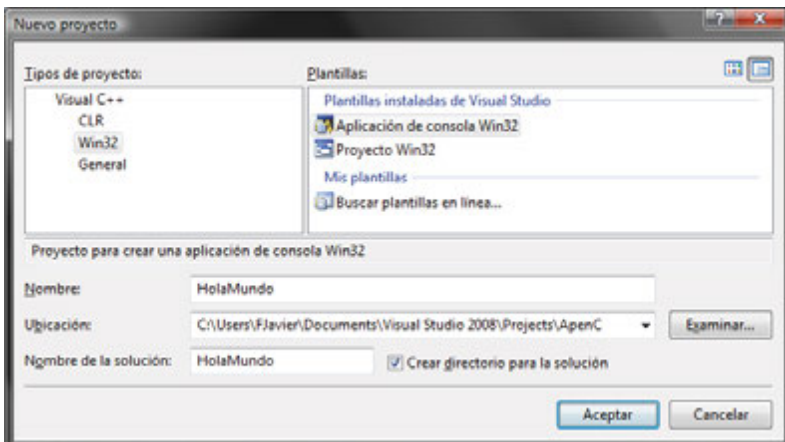
Microsoft Visual C++

En la figura siguiente se puede observar la página de inicio del entorno de desarrollo integrado *Microsoft Visual C++*.

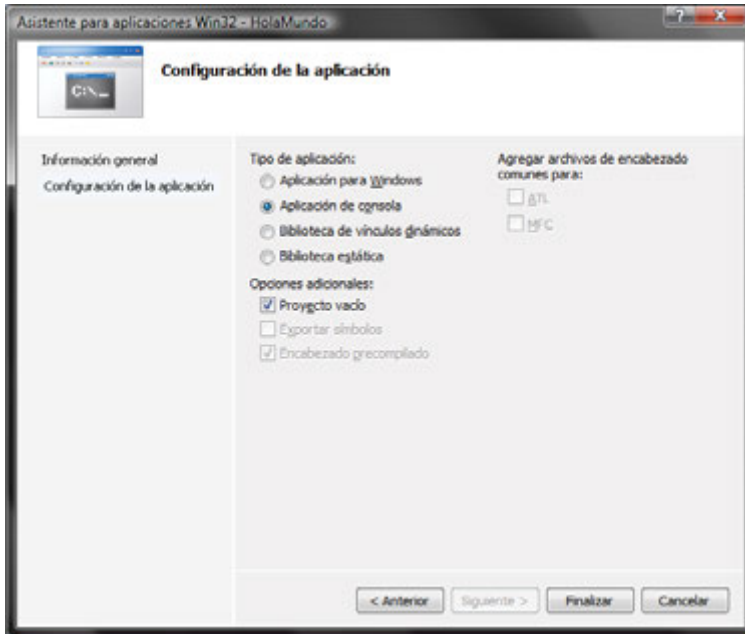


Para editar y ejecutar el programa *HolaMundo.cpp* expuesto en el capítulo 1 utilizando este entorno de desarrollo, los pasos a seguir son los siguientes:

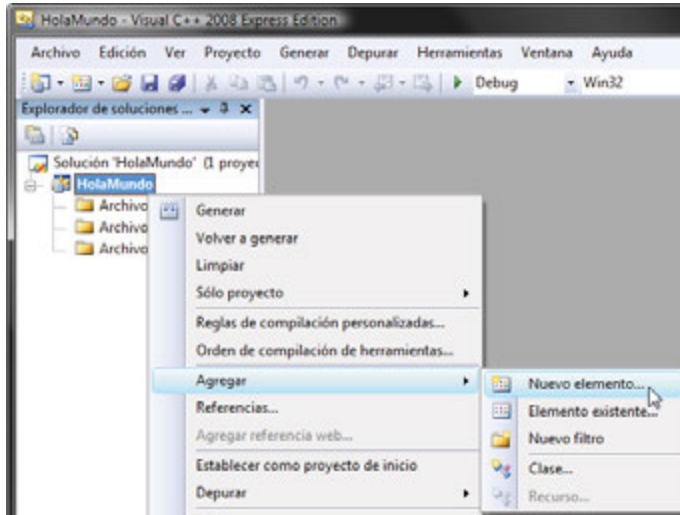
1. Partiendo de la página de inicio de *Visual C++*, hacemos clic en el botón *Crear proyecto* para crear un proyecto nuevo o bien ejecutamos la orden *Archivo > Nuevo > Proyecto*. Esta acción hará que se visualice una ventana que mostrará en su panel izquierdo los tipos de proyectos que se pueden crear, y en su panel derecho las plantillas que se pueden utilizar; la elección de una o de otra dependerá del tipo de aplicación que deseemos construir. La figura siguiente muestra esta ventana:



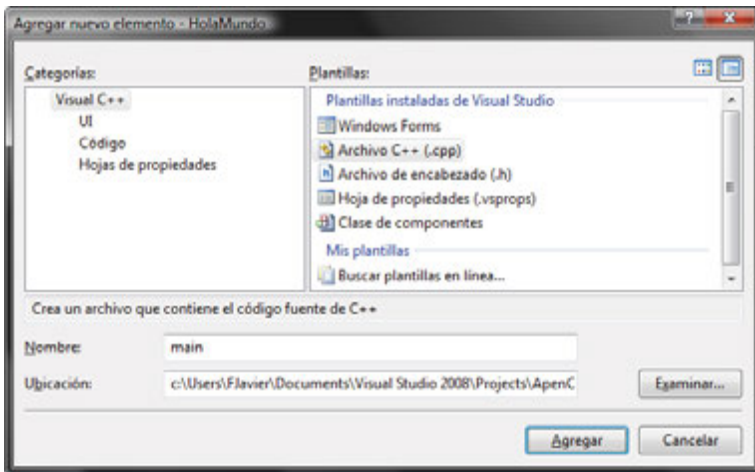
- Para nuestro ejemplo, elegimos el tipo de proyecto *Visual C++ Win32* y la plantilla *Aplicación de consola Win32*. Después especificamos el nombre del proyecto y su ubicación; observe que el proyecto será creado en una carpeta con el mismo nombre. A continuación pulsamos el botón *Aceptar*. Esta acción visualizará la ventana mostrada en la figura siguiente, que permitirá establecer las opciones necesarias para generar una aplicación de consola partiendo de un proyecto vacío:



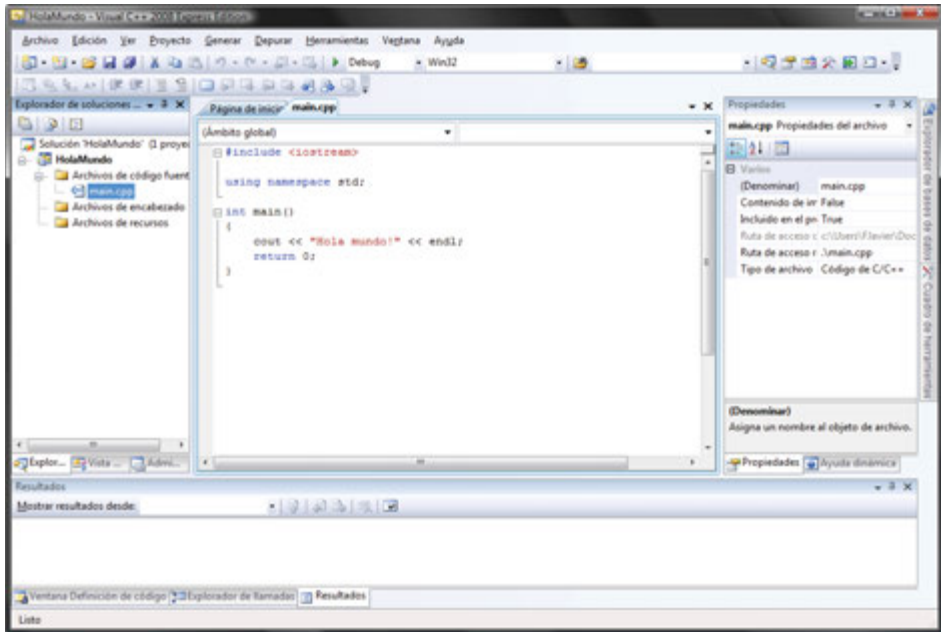
- Una vez configurada la aplicación, pulsamos el botón *Finalizar*. El resultado será un proyecto vacío al que podremos añadir ficheros. Por ejemplo, para añadir el fichero *HolaMundo.cpp*, hacemos clic con el botón derecho del ratón sobre el nombre del proyecto y seleccionamos la orden *Agregar > Nuevo elemento...*



4. La acción ejecutada en el punto anterior muestra la ventana que se muestra a continuación, la cual nos permitirá elegir la plantilla para el fichero, en nuestro caso *Archivo C++ (.cpp)*, y especificar el nombre y la ubicación del mismo.



5. El siguiente paso es escribir el código que se almacenará en este fichero, según muestra la figura siguiente:



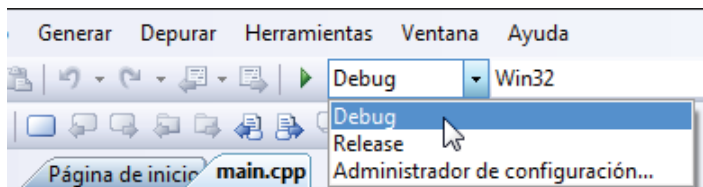
En esta figura observamos una ventana principal que contiene otras ventanas con varias páginas cada una de ellas. La que está en la parte central está mostrando la página de edición del fichero *main.cpp* que estamos editando y tiene oculta la página de inicio. La que está en la parte izquierda está mostrando el explorador de soluciones; éste lista el nombre de la solución (una solución puede contener uno o más proyectos), el nombre del proyecto o proyectos y el nombre de los ficheros que componen el proyecto; en nuestro caso sólo tenemos el fichero *main.cpp* donde escribiremos el código de las acciones que tiene que llevar a cabo nuestra aplicación; el explorador de soluciones oculta la vista de clases, cuya misión es mostrar el conjunto de clases que forman una aplicación orientada a objetos; haga clic en la pestaña *Vista de clases* para observar su contenido. La ventana que hay en la parte derecha muestra la página de propiedades y oculta la página correspondiente a la ayuda dinámica; haga clic en la pestaña *Ayuda dinámica* si quiere consultar la ayuda relacionada con la selección que haya realizado en la página de edición. Y la ventana que hay debajo de la página de edición puede mostrar varias páginas, por ejemplo, la de resultados de la compilación.

6. Una vez editado el programa, para compilarlo ejecutamos la orden *Generar ...* del menú *Generar* y para ejecutarlo, seleccionamos la orden *Iniciar sin depurar* del menú *Depurar* o bien pulsamos las teclas *Ctrl+F5*.

DEPURAR LA APLICACIÓN

¿Por qué se depura una aplicación? Porque los resultados que estamos obteniendo con la misma no son correctos y no sabemos por qué. El proceso de depuración consiste en ejecutar la aplicación paso a paso, indistintamente por sentencias o por funciones, con el fin de observar el flujo seguido durante su ejecución, así como los resultados intermedios que se van sucediendo, con la finalidad de detectar las anomalías que producen un resultado final erróneo. Para llevarlo a cabo es preciso compilar la aplicación indicando que va a ser depurada; de esta forma, el compilador añadirá el código que permitirá este proceso.

Hay dos configuraciones bajo las que se puede compilar una aplicación: *Release* y *Debug*. La primera permite obtener un programa ejecutable optimizado en código y en velocidad, y la segunda, un programa ejecutable con código extra necesario para depurar la aplicación.



Por ejemplo, para depurar una aplicación utilizando el depurador del entorno de desarrollo de *Visual C++*, debe activar la configuración *Debug* antes de iniciar su compilación. Para ello, proceda como muestra la figura anterior.

Una vez construida la aplicación bajo la configuración *Debug* podrá, si lo necesita, depurar la misma. Para ello, ejecute la orden *Depurar > Paso por instrucciones* y utilice las órdenes del menú *Depurar* o los botones correspondientes de la barra de herramientas (para saber el significado de cada botón, ponga el puntero del ratón sobre cada uno de ellos).



De forma resumida, las órdenes disponibles para depurar una aplicación son las siguientes:

- *Continuar* o *F5*. Continúa la ejecución de la aplicación en modo depuración hasta encontrar un punto de parada o hasta el final si no hay puntos de parada.
- *Interrumpir todos*. El depurador detendrá la ejecución de todos los programas que se ejecutan bajo su control.

- *Detener depuración* o *Mayús+F5*. Detiene el proceso de depuración.
- *Reiniciar* o *Ctrl+Mayús+F5*. Reinicia la ejecución de la aplicación en modo depuración.
- *Mostrar la instrucción siguiente*. Muestra la siguiente instrucción a ejecutar.
- *Paso a paso por instrucciones* o *F11*. Ejecuta la aplicación paso a paso. Si la línea a ejecutar coincide con una llamada a una función definida por el usuario, dicha función también se ejecuta paso a paso.
- *Paso a paso por procedimientos* o *F10*. Ejecuta la aplicación paso a paso. Si la línea a ejecutar coincide con una llamada a una función definida por el usuario, dicha función no se ejecuta paso a paso, sino de una sola vez.
- *Paso a paso para salir* o *Mayús+F11*. Cuando una función definida por el usuario ha sido invocada para ejecutarse paso a paso, utilizando esta orden se puede finalizar su ejecución en un solo paso.
- *Insertar/Quitar punto de interrupción* o *F9*. Pone o quita un punto de parada en la línea sobre la que está el punto de inserción.
- *Ejecutar hasta el cursor* o *Ctrl+F10*. Ejecuta el código que hay entre la última línea ejecutada y la línea donde se encuentra el punto de inserción.
- *Inspección rápida* o *Ctrl+Alt+Q*. Visualiza el valor de la variable que está bajo el punto de inserción o el valor de la expresión seleccionada (sombreada).

Para ejecutar la aplicación en un solo paso, seleccione la orden *Iniciar sin depurar* (*Ctrl+F5*) del menú *Depurar*.

Con otro entorno integrado de desarrollo, por ejemplo *CodeBlocks*, los pasos a seguir para depurar una aplicación son similares.

MICROSOFT C++: INTERFAZ DE LÍNEA DE ÓRDENES

Los ficheros que componen una aplicación C++ pueden ser escritos utilizando cualquier editor de texto ASCII; por ejemplo, el *Bloc de notas*. Una vez editados y guardados todos los ficheros que componen la aplicación, el siguiente paso es compilarlos y enlazarlos para obtener el fichero ejecutable correspondiente a la misma. La orden para realizar estas operaciones utilizando la implementación *Microsoft C++* es la siguiente:

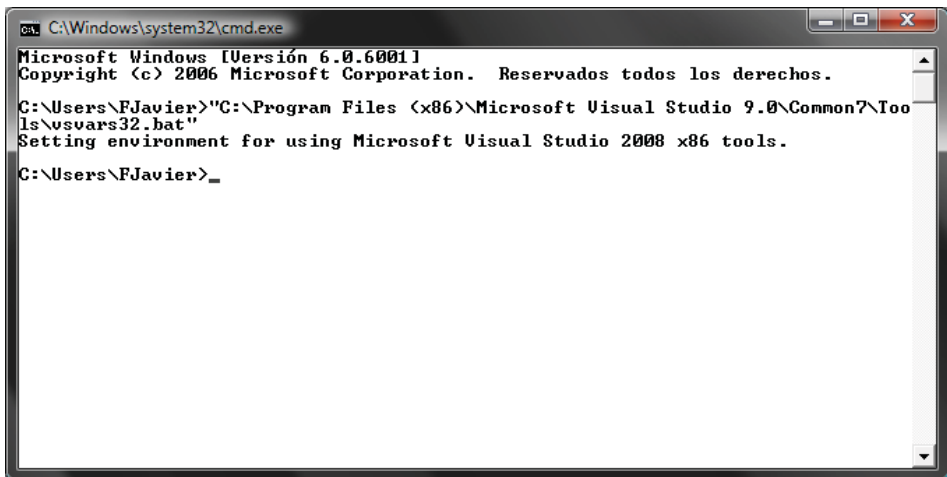
```
cl fichero01.cpp [fichero02 [fichero03] ...]
```

El nombre del fichero ejecutable resultante será el mismo que el nombre del primer fichero especificado, pero con extensión *.exe*.

Previamente, para que el sistema operativo encuentre la utilidad *cl*, los ficheros de cabecera (directriz **include**) y las bibliotecas dinámicas y estáticas, cuando son invocados desde la línea de órdenes, hay que definir en el entorno de trabajo las siguientes variables:

```
set path=%path%;ruta de los ficheros .exe y .dll
set include=ruta de los ficheros .h
set lib=ruta de los ficheros .lib
```

La expresión *%path%* representa el valor actual de la variable de entorno *path*. Una ruta va separada de la anterior por un punto y coma. Estas variables también pueden ser establecidas ejecutando el fichero *vcvars32.bat* que aporta *Visual C++*. En la figura siguiente puede observarse un ejemplo:



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. Reservados todos los derechos.

C:\Users\FJavier>"C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\Tools\
ls\vcvars32.bat"
Setting environment for using Microsoft Visual Studio 2008 x86 tools.

C:\Users\FJavier>_
```

Una vez establecidas estas variables, ya puede invocar al compilador C++ y al enlazador. En la figura siguiente se puede observar, como ejemplo, el proceso seguido para compilar *main.cpp*:

```

C:\Windows\system32\cmd.exe

C:\Users\FJavier>cd C:\Users\FJavier\Documents\ejemplos\ApenC\HolaMundo

C:\Users\FJavier\Documents\ejemplos\ApenC\HolaMundo>cl main.cpp
Compilador de optimización de C/C++ de 32 bits de Microsoft (R) versión 15.00.30729.01 para 80x86
Copyright (C) Microsoft Corporation. All rights reserved.

main.cpp
C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\INCLUDE\xlocale(342) : warning C4530: Se ha utilizado el controlador de excepciones de C++, pero la semántica de desenredo no está habilitada. Especifique /EHsc
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.

/out:main.exe
main.obj

C:\Users\FJavier\Documents\ejemplos\ApenC\HolaMundo>main
Hola mundo!

C:\Users\FJavier\Documents\ejemplos\ApenC\HolaMundo>_

```

Observe que antes de invocar al compilador hemos cambiado al directorio de la aplicación (*cd*). Después invocamos al compilador C++ (*cl*). El resultado es *main.exe*. Para ejecutar este fichero, escriba *main* en la línea de órdenes y pulse *Entrar*.

LINUX: INTERFAZ DE LÍNEA DE ÓRDENES

Los ficheros que componen una aplicación C++ realizada bajo GNU-Linux pueden ser escritos utilizando cualquier editor de texto ASCII proporcionado por éste. Una vez editados y guardados todos los ficheros que componen la aplicación, el siguiente paso es compilarlos y enlazarlos para obtener el fichero ejecutable correspondiente a la misma. La orden para realizar estas operaciones es la siguiente:

```
g++ fichero01.cpp [fichero02 [fichero03] ...] -o fichero_ejecutable
```

En el caso de Linux, las rutas de acceso para que el sistema operativo encuentre la utilidad *g++*, los ficheros de cabecera y las bibliotecas, cuando son invocados desde la línea de órdenes, ya están definidas en el entorno de trabajo.

En la figura siguiente se puede observar, como ejemplo, el proceso seguido para compilar *HolaMundo.cpp*:

```

linux@linux:~/...ts/HolaMundo/ApenC
Archivo Editar Ver Terminal Solapas Ayuda
linux@linux:~> PATH=$PATH: .
linux@linux:~> cd projects/HolaMundo/ApenC/
linux@linux:~/projects/HolaMundo/ApenC> ls -l
total 4
-rw-r--r-- 1 linux users 97 ene 19 17:08 main.cpp
linux@linux:~/projects/HolaMundo/ApenC> g++ -o main main.cpp
linux@linux:~/projects/HolaMundo/ApenC> ls -l
total 16
-rwxr-xr-x 1 linux users 9866 ene 19 17:33 main
-rw-r--r-- 1 linux users 97 ene 19 17:08 main.cpp
linux@linux:~/projects/HolaMundo/ApenC> main
Hola mundo!
linux@linux:~/projects/HolaMundo/ApenC> █

```

Observe que primero hemos cambiado al directorio de la aplicación (*cd*), después hemos visualizado el contenido de ese directorio (*ls -l*) y finalmente hemos invocado al compilador C++ (*g++*). El fichero ejecutable resultante es el especificado por la opción *-o*, en el ejemplo *main*, o *a.out* por omisión.

Para ejecutar la aplicación del ejemplo, escriba *main* en la línea de órdenes y pulse *Entrar*. Si al realizar esta operación se encuentra con que no puede hacerlo porque el sistema no encuentra el fichero especificado, tiene que añadir la ruta del directorio actual de trabajo a la variable de entorno *PATH*. Esto se hace así:

```
PATH=$PATH: .
```

La expresión *\$PATH* representa el valor actual de la variable de entorno *PATH*. Una ruta va separada de la anterior por dos puntos. El directorio actual está representado por el carácter punto.

El depurador gdb de GNU

Cuando se tiene la intención de depurar un programa C escrito bajo GNU, en el momento de compilarlo se debe especificar la opción *-g* o *-g3*. Esta opción indica al compilador que incluya información extra para el depurador en el fichero objeto. Por ejemplo:

```
g++ -g3 prog01.cpp -o prog01.exe
```

La orden anterior compila y enlaza el fichero fuente *prog01.cpp*. El resultado es un fichero ejecutable *prog01.exe* con información para el depurador.

Una vez compilado un programa con las opciones necesarias para depurarlo, invocaremos a *gdb* para proceder a su depuración. La sintaxis es la siguiente:

gdb fichero-ejecutable

El siguiente ejemplo invoca al depurador *gdb* de GNU-Linux, que carga el fichero ejecutable *prog01* en memoria para depurarlo.

```
gdb prog01.exe
```

Una vez que se ha invocado el depurador, desde la línea de órdenes se pueden ejecutar órdenes como las siguientes:

- *break [fichero:]función*. Establece un punto de parada en la función indicada del fichero especificado. Por ejemplo, la siguiente orden pone un punto de parada en la función *escribir*.

```
b escribir
```

- *break [fichero:]línea*. Establece un punto de parada en la línea indicada. Por ejemplo, la siguiente orden pone un punto de parada en la línea 10.

```
b 10
```

- *delete punto-de-parada*. Elimina el punto de parada especificado. Por ejemplo, la siguiente orden elimina el punto de parada 1 (primero).

```
d 1
```

- *run [argumentos]*. Inicia la ejecución de la aplicación que deseamos depurar. La ejecución se detiene al encontrar un punto de parada o al finalizar la aplicación. Por ejemplo:

```
run
```

- *print expresión*. Visualiza el valor de una variable o de una expresión. Por ejemplo, la siguiente orden visualiza el valor de la variable *total*.

```
p total
```

- *next*. Ejecuta la línea siguiente. Si la línea coincide con una llamada a una función definida por el usuario, no se entra a depurar la función. Por ejemplo:

n

- *continue*. Continúa con la ejecución de la aplicación. Por ejemplo:

c

- *step*. Ejecuta la línea siguiente. Si la línea coincide con una llamada a una función definida por el usuario, se entra a depurar la función. Por ejemplo:

s

- *list*. Visualiza el código fuente. Por ejemplo:

l

- *bt*. Visualiza el estado de la pila de llamadas en curso (las llamadas a funciones).
- *help [orden]*. Solicita ayuda sobre la orden especificada.
- *quit*. Finaliza el trabajo de depuración.

INSTALACIÓN DEL PAQUETE DE DESARROLLO

En el apéndice *Entornos de desarrollo* hemos visto cómo escribir y ejecutar una aplicación C++ desde dos entornos de desarrollo diferentes: *CodeBlocks*, que incluye un compilador C/C++ de *GCC (GNU Compiler Collection)*, y *Microsoft Visual Studio* (o bien *Microsoft Visual C++ Express*), que incluye el compilador Microsoft C/C++. También hemos visto que podemos hacerlo de dos formas diferentes: editando, compilando y depurando desde el entorno de desarrollo, o bien desde la línea de órdenes. Veamos a continuación cómo instalar estos compiladores en una plataforma Windows (Windows 2000/XP/Vista).

INSTALACIÓN DE MinGW

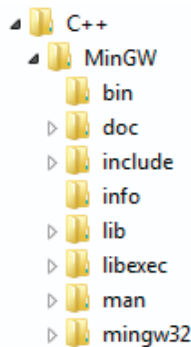
MinGW (Minimalist GNU for Win32) es un paquete que proporciona una versión nativa de Win32 de GCC (*gcc*, *g++*, *g77*, etc.), el depurador *gdb*, *make*, *win32api*, y otras utilidades. Se puede realizar una instalación personalizada instalando por una parte la implementación GCC, y por otra el entorno de desarrollo integrado (EDI) *CodeBlocks*, o bien se puede instalar una versión de *CodeBlocks* que ya incluye *MinGW*. En nuestro caso vamos a instalar la implementación *MinGW* y el entorno integrado *CodeBlocks* por separado. De esta forma podrá instalar otros EDI como *Eclipse* o *NetBeans* que necesitan de GCC.

Para realizar la instalación descargue el fichero *MinGW-x.x.x.exe*, o bien utilice la versión suministrada en el CD del libro y ejecútelo. Después, siguiendo los pasos especificados por el programa de instalación, seleccione *descargar e instalar*, acepte el acuerdo de licencia, elija la versión que quiere descargar (se recomienda descargar la *actual*), seleccione los componentes que desea instalar (al

menos, como muestra la figura siguiente, *MinGW* y *g++*), seleccione la carpeta donde lo quiere instalar y proceda a la descarga e instalación.



La figura siguiente muestra un ejemplo de instalación:



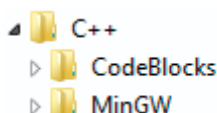
Observe si en la carpeta *bin* están las utilidades *gdb.exe* (depurador) y *make.exe* o *mingw32-make.exe* (para la construcción de proyectos). Si no están, descárguelos e instálelos.

Esta instalación le permitirá editar, compilar, ejecutar y depurar sus programas C++ desde una ventana de consola. Para ello, una vez abierta la ventana debe establecer la siguiente variable de entorno:

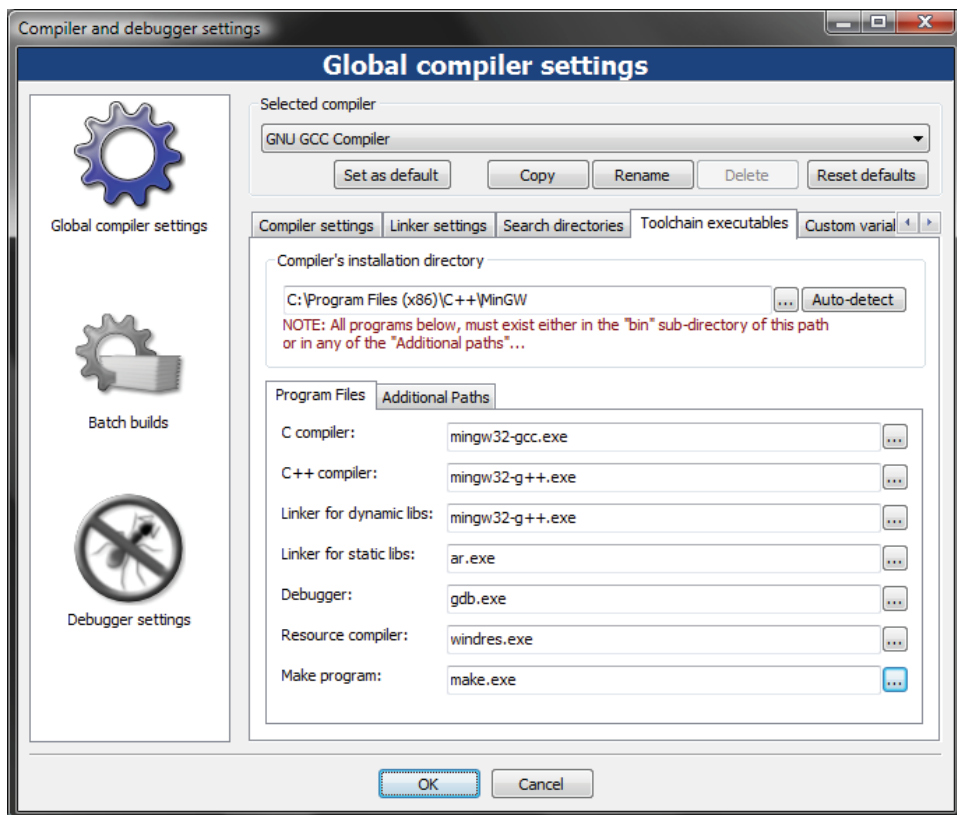
```
SET PATH=%PATH%;C:\...\MinGW\bin
```

Instalación de CodeBlocks

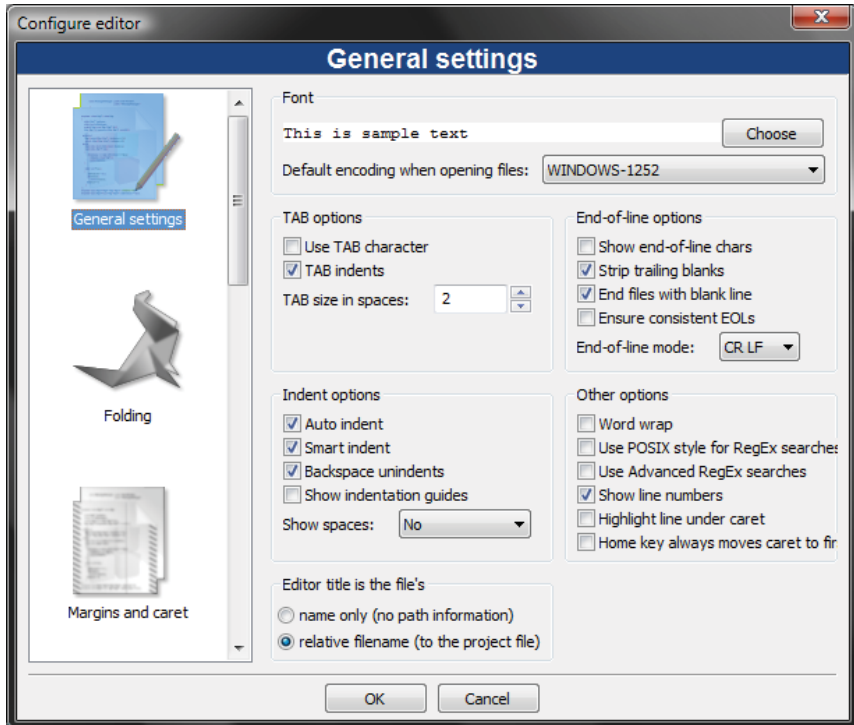
Una vez instalado el entorno de desarrollo de C++ puede instalar un entorno de desarrollo integrado (EDI) que integre el editor soportado por él y el compilador y el depurador anteriormente instalados. Para ello, ejecute el fichero *codeblocks-8.02-setup.exe* localizado en la carpeta *EDI\MinGW+CodeBlocks* del CD del libro, o bien descárguelo de Internet, e instálelo, según la figura anterior, en la carpeta C++. El resultado será similar al presentado por la figura siguiente:



A continuación abra el EDI, seleccione en la orden *Compiler and debugger...* del menú *Settings*, haga clic en la pestaña *Toolchain executables* de la ventana que se visualiza y verifique que la ruta de *MinGW* es la especificada y que las utilidades seleccionadas están en la carpeta *MinGW\bin*.



Finalmente, personalice la instalación a su medida a través de las órdenes *Editor...* y *Environment...* del menú *Settings*. Por ejemplo, active la casilla de verificación *Show line numbers* si quiere mostrar los números de la líneas del programa.



Estas dos instalaciones que acabamos de realizar pueden ser hechas de una sola vez ejecutando el fichero *codeblocks-8.02mingw-setup.exe* localizado en la carpeta *EDI\CodeBlocks* del CD del libro, o bien descargándolo de Internet, e instalándolo en la carpeta deseada. Esta forma de proceder es menos versátil que la anterior, ya que no permite actualizar los paquetes *MinGW* y *CodeBlocks* de forma independiente y tampoco deja *MinGW* a disposición de otros EDI.

INSTALACIÓN DE Microsoft C/C++

También, si lo prefiere puede utilizar el compilador C/C++ de Microsoft. Para ello, tiene que instalar el paquete *Microsoft Visual Studio* o *Microsoft Visual C++ Express Edition* (esta última versión puede descargarla de forma gratuita de Internet), ya que el kit de desarrollo de software (*.Net Framework SDK*) de Microsoft sólo incluye la colección de compiladores C# y Visual Basic.

CÓDIGOS DE CARACTERES

UTILIZACIÓN DE CARACTERES ANSI CON WINDOWS

Una tabla de códigos es un juego de caracteres donde cada uno tiene asignado un número utilizado para su representación interna. Visual Basic utiliza Unicode para almacenar y manipular cadenas, pero también puede manipular caracteres en otros códigos como ANSI o ASCII.

ANSI (*American National Standards Institute*) es el juego de caracteres estándar más utilizado por los equipos personales. Como el estándar ANSI sólo utiliza un byte para representar un carácter, está limitado a un máximo de 256 caracteres. Aunque es adecuado para el inglés, no acepta totalmente otros idiomas. Para escribir un carácter ANSI que no esté en el teclado:

1. Localice en la tabla que se muestra en la página siguiente el carácter ANSI que necesite y observe su código numérico.
2. Pulse la tecla *Bloq Núm* (Num Lock) para activar el teclado numérico.
3. Mantenga pulsada la tecla *Alt* y utilice el teclado numérico para pulsar el 0 y a continuación las teclas correspondientes al código del carácter.

Por ejemplo, para escribir el carácter \pm en el entorno Windows, mantenga pulsada la tecla *Alt* mientras escribe 0177 en el teclado numérico. Pruebe en la consola del sistema (línea de órdenes).

Los 128 primeros caracteres (códigos 0 a 127) son los mismos en las tablas de códigos ANSI, ASCII y Unicode.

JUEGO DE CARACTERES ANSI

DEC	CAR	DEC	CAR	DEC	CAR	DEC	CAR
33	!	89	Y	146	'	202	Ê
34	"	90	Z	147	..	203	Ë
35	#	91	[148	"	204	Ì
36	\$	92	\	149	o	205	Í
37	%	93]	150	-	206	Î
38	&	94	^	151	-	207	Ï
39	'	96	~	152	☒	208	Ð
40	(97	a	153	☒	209	Ñ
41)	98	b	154	☒	210	Ò
42	*	99	c	155	☒	211	Ó
43	+	100	d	156	☒	212	Ô
44	,	101	e	157	☒	213	Õ
45	-	102	f	157	☒	214	Ö
46	.	103	g	159	☒	215	×
47	/	104	h	160		216	Ø
48	0	105	i	161	;	217	Ù
49	1	106	j	162	c	218	Ú
50	2	107	k	163	£	219	Û
51	3	108	l	164	¤	220	Ü
52	4	109	m	165	¥	221	Ý
53	5	110	n	166		222	Þ
54	6	111	o	167	§	223	ß
55	7	112	p	168	"	224	à
56	8	113	q	169	e	225	á
57	9	114	r	170	*	226	â
58	:	115	s	171	-	227	ã
59	;	116	t	172	-	228	ä
60	<	117	u	173	-	229	å
61	=	118	v	174	*	230	æ
62	>	119	w	175	-	231	ç
63	?	120	x	176	°	232	è
64	@	121	y	177	±	233	é
65	A	122	z	178	²	234	ê
66	B	123	{	179	³	235	ë
67	C	124		180	´	236	ì
68	D	125	}	181	µ	237	í
69	E	126	~	182	¶	238	î
70	F	127	☒	183	·	239	ï
71	G	128	☒	184	.	240	ð
72	H	129	☒	185	i	241	ñ
73	I	130	☒	186	°	242	ò
74	J	131	☒	187	"	243	ó
75	K	132	☒	188	¼	244	ô
76	L	133	☒	189	½	245	õ
77	M	134	☒	190	¾	246	ö
78	N	135	☒	191	¿	247	÷
79	O	136	☒	192	À	248	ø
80	P	137	☒	193	Á	249	ù
81	Q	138	☒	194	Â	250	ú
82	R	139	☒	195	Ã	251	û
83	S	140	☒	196	Ä	252	ü
84	T	141	☒	197	Å	253	ý
85	U	142	☒	198	Æ	254	þ
86	V	143	☒	199	Ç	255	ÿ
87	W	144	☒	200	È		
88	X	145	.	201	É		

UTILIZACIÓN DE CARACTERES ASCII

En MS-DOS y fuera del entorno Windows se utiliza el juego de caracteres ASCII. Para escribir un carácter ASCII que no esté en el teclado:

1. Busque el carácter en la tabla de códigos que coincida con la tabla activa. Utilice la orden **chcp** para saber qué tabla de códigos está activa.
2. Mantenga pulsada la tecla *Alt* y utilice el teclado numérico para pulsar las teclas correspondientes al número del carácter que desee.

Por ejemplo, si está utilizando la tabla de códigos 850, para escribir el carácter π mantenga pulsada la tecla *Alt* mientras escribe 227 en el teclado numérico.

JUEGO DE CARACTERES ASCII

VALOR DECIMAL	VALOR HEXA-DECIMAL	CONTROL CARACT.	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.	VALOR DECIMAL	VALOR HEXA-DECIMAL	CARACT.
000	00	NUL		043	2B	+	086	56	V	129	81	ù	172	AC	¼	215	D7	#
001	01	SOH	☺	044	2C	,	087	57	W	130	82	é	173	AD	í	216	D8	≠
002	02	STX	☹	045	2D	.	088	58	X	131	83	ð	174	AE	ª	217	D9	┘
003	03	ETX	♥	046	2E	-	089	59	Y	132	84	ã	175	AF	«	218	DA	┘
004	04	EOT	♦	047	2F	/	090	5A	Z	133	85	ä	176	BO	»	219	DB	■
005	05	ENQ	♣	048	30	0	091	5B	[134	86	å	177	B1	▣	220	DC	■
006	06	ACK	♠	049	31	1	092	5C	\	135	87	æ	178	B2	▤	221	DD	■
007	07	BEL	•	050	32	2	093	5D]	136	88	ç	179	B3	▥	222	DE	■
008	08	BS	■	051	33	3	094	5E	^	137	89	à	180	B4	▧	223	DF	■
009	09	HT	○	052	34	4	095	5F	_	138	8A	á	181	B5	▨	224	EO	α
010	0A	LF	☐	053	35	5	096	60	`	139	8B	â	182	B6	▩	225	E1	β
011	0B	VT	♂	054	36	6	097	61	a	140	8C	ä	183	B7	▪	226	E2	γ
012	0C	FF	♀	055	37	7	098	62	b	141	8D	å	184	B8	▫	227	E3	π
013	0D	CR	♪	056	38	8	099	63	c	142	8E	ä	185	B9	▬	228	E4	∑
014	0E	SO	♫	057	39	9	100	64	d	143	8F	å	186	BA	▭	229	E5	σ
015	0F	SI	⊙	058	3A	:	101	65	e	144	90	é	187	BB	▮	230	E6	μ
016	10	DLE	▶	059	3B	;	102	66	f	145	91	æ	188	BC	▯	231	E7	τ
017	11	DC1	◀	060	3C	<	103	67	g	146	92	ç	189	BD	▰	232	E8	φ
018	12	DC2	‡	061	3D	=	104	68	h	147	93	ä	190	BE	▱	233	E9	⊖
019	13	DC3		062	3E	>	105	69	i	148	94	å	191	BF	▲	234	EA	⊗
020	14	DC4	†	063	3F	?	106	6A	j	149	95	ä	192	C0	△	235	EB	⊘
021	15	NAK	§	064	40	@	107	6B	k	150	96	ú	193	C1	▴	236	EC	∞
022	16	SYN	▬	065	41	A	108	6C	l	151	97	û	194	C2	▵	237	ED	⊙
023	17	ETB	↓	066	42	B	109	6D	m	152	98	ÿ	195	C3	▶	238	EE	€
024	18	CAN		067	43	C	110	6E	n	153	99	ô	196	C4	▷	239	EF	∩
025	19	EM		068	44	D	111	6F	o	154	9A	Û	197	C5	▸	240	FO	≡
026	1A	SUB	—	069	45	E	112	70	p	155	9B	ç	198	C6	▹	241	F1	±
027	1B	ESC	—	070	46	F	113	71	q	156	9C	è	199	C7	►	242	F2	≥
028	1C	FS	└	071	47	G	114	72	r	157	9D	ÿ	200	C8	▻	243	F3	≤
029	1D	GS	↔	072	48	H	115	73	s	158	9E	ÿ	201	C9	▼	244	F4	
030	1E	RS	▲	073	49	I	116	74	t	159	9F	f	202	CA	▽	245	F5	j
031	1F	US	▼	074	4A	J	117	75	u	160	A0	á	203	CB	▾	246	F6	→
032	20	SP	Space	075	4B	K	118	76	v	161	A1	í	204	CC	▿	247	F7	≈
033	21			076	4C	L	119	77	w	162	A2	ä	205	CD	▹	248	F8	°
034	22		*	077	4D	M	120	78	x	163	A3	ó	206	CE	▸	249	F9	•
035	23		#	078	4E	N	121	79	y	164	A4	ñ	207	CF	▹	250	FA	.
036	24		\$	079	4F	O	122	7A	z	165	A5	ñ	208	D0	▹	251	FB	√
037	25		%	080	50	P	123	7B	{	166	A6	°	209	D1	▹	252	FC	∩
038	26		&	081	51	Q	124	7C		167	A7	°	210	D2	▹	253	FD	'
039	27		'	082	52	R	125	7D	}	168	A8	¿	211	D3	▹	254	FE	•
040	28		{	083	53	S	126	7E	-	169	A9	▹	212	D4	▹	255	FF	
041	29		}	084	54	T	127	7F	⊖	170	AA	▹	213	D5	▹			
042	2A		*	085	55	U	128	80	Ç	171	AB	½	214	D6	▹			

JUEGO DE CARACTERES UNICODE

UNICODE es un juego de caracteres en el que se emplean 2 bytes (16 bits) para representar cada carácter. Esto permite la representación de cualquier carácter en cualquier lenguaje escrito en el mundo, incluyendo los símbolos del chino, japonés o coreano.

Códigos Unicode de los dígitos utilizados en español:

\u0030-\u0039 0-9 ISO-LATIN-1

Códigos Unicode de las letras y otros caracteres utilizados en español:

\u0024	\$ signo dólar
\u0041-\u005a	A-Z
\u005f	_
\u0061-\u007a	a-z
\u00c0-\u00d6	À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï Ð Ñ Ò Ó Ô Õ Ö
\u00d8-\u00f6	Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö
\u00f8-\u00ff	ø ù ú û ü ý þ ÿ

Dos caracteres son idénticos sólo si tienen el mismo código Unicode.

ÍNDICE

#

#define, 64, 344
#endif, 344
#if, 344
#include, 64, 341

A

abort, 635
abs, 851
acceso a los elementos, 173
acceso a un dato miembro, 384
acceso aleatorio, 698
acceso secuencial, 687, 688
acos, 849
adjustfield, 108
adquisición de recursos, 643
agregación, 380
algorithm, 175, 201
algoritmo Boyer y Moore, 806
ambigüedades, 313
amistad, 398
ancho, 840
and, 41
and_eq, 44
anidar if, 129
anidar while, do, o for, 140
ANSI, 883
añadir un elemento a una matriz, 392
aplicación, implementación, 351
append, 188
árbol, 756
 binario, 757
 binario de búsqueda, 761
 binario perfectamente equilibrado, 774
 árbol, recorrer, 759
archivo, 662
área de escritura, 670
área de lectura, 670
argumento, 67
argumentos
 en la línea de órdenes, 301
 pasados a una función, 22
 pasados por valor, 73
 pasar a una función, 73
 por referencia, 74
aritmética de punteros, 229
ASCII, 100, 883, 885
 valor, 113
asignación, 15
 compuesta, 431
 de objetos, 362
 dinámica de memoria, 248
asignar bytes desde una dirección, 228
asin, 849
assign, 186
at, 173
atan, 849
atan2, 850
ate, 672
atof, 847
atoi, 848
atol, 848
atributo static, 375
atributo, iniciar, 337
atributos, 337
 con el mismo nombre, 496
auto, 83
auto_ptr, 649

B

back, 174
 bad, 105
 bad_alloc, 250, 625
 bad_cast, 521, 625
 bad_exception, 625, 636
 bad_typeid, 535, 625
 basefield, 108
 basic_ios, 107
 basic_istream, 107
 basic_ostream, 107
 begin, 174
 biblioteca de C, 837
 biblioteca estándar de C++, 9, 89, 833
 bin, 6
 binario, formato/modo, 686
 binary_search, 810
 bit, 3
 bitand, 43
 bitor, 43
 bits de error, 859
 bitset, 786
 bloque, 13, 38, 66
 bloque de código, 7
 bool, 29
 boolalpha, 108
 borrar los elementos de una lista, 718
 borrar nodo, 768
 borrar un elemento de una lista, 717
 Boyer y Moore, 806
 break, 136, 149
 búfer, 664
 búfer de entrada, limpiar, 116
 burbuja, 798
 buscar nodo, 766
 buscar un elemento en una lista, 715
 buscar un elemento en una matriz, 393
 búsqueda binaria, 805
 búsqueda de cadenas, 806
 búsqueda secuencial, 805
 byte, 3

C

c_str, 187
 C++0x, 815
 cadena de caracteres, leer y escribir, 184
 cadena, principio y final, 239
 cadenas de caracteres, 183, 472
 calificadores, 83
 campo, 662

capacity, 190
 capturar cualquier excepción, 632
 capturar la excepción, 630
 carácter \n, 115
 caracteres de C++, 26
 caracteres, leer, 112
 caracteres, manipular, 178
 CArbolBinE, 783
 cast, 55
 catch, 152, 630
 CCola, 741
 ceil, 851
 cerr, 99, 102
 char, 14
 char_traits::eof, 677
 cin, 101
 clase, 32, 335
 abstracta, 551
 amiga de otra clase, 400
 bad_alloc, 627
 base, 483
 base directa, 512
 base indirecta, 512
 base privada, 490
 base protegida, 490
 base pública, 490
 base replicada, 553
 base virtual, 556
 bien formada, 560
 CArbolBinB, 771
 CCadena, 472
 CComplejo, 464
 contenedor, 579
 CRacional, 428
 derivada, 32, 483
 derivada, control de acceso, 489
 derivada, definir, 489
 exception, 626
 filebuf, 671
 fstream, 683
 genérica, 722
 genérica, 594
 ifstream, 681
 interna, 382
 ios, 98
 iostream, 98
 iostream, 678
 istream, 98
 istream, 675
 lista lineal simplemente enlazada, 719
 ofstream, 679
 ostream, 98

- clase
 - ostream, 673
 - runtime_error, 626
 - streambuf, 670
 - string, 185
 - virtual, conversiones, 563
 - wstring, 185
- clases con ficheros, 690
- clases derivadas
 - constructor copia, 503
 - conversiones, 505
 - funciones amigas, 514
 - miembros static, 514
 - operador de asignación, 503
- clases en ficheros de cabecera, 344
- clases hermanas, 553
- class, 335, 580
- clear, 106, 114, 175, 626
- clearerr, 859
- CListaCircularDE, 747
- CListaCircularSE, 734
- CListaLinealSE, 723
- CListaLinealSEO, 789
- clock, 853
- clog, 99, 102
- clonar, 529, 541
- close
 - filebuf, 672
 - fstream, 684
 - ifstream, 682
 - ofstream, 679
- cola, 741
- comentario, 16
- compilador, 3
- compilar un programa, 8
 - formado por varios ficheros, 345
- complejos, 463
- const, 37, 354, 732
- const_cast, 55, 355
- constante de carácter, 18
- constante simbólica, 37
- constructor, 357, 358
 - copia, 363
 - por omisión, 504
 - redefinir, 550
 - virtual, 529
 - de la clase base, invocar, 500
 - de la clase derivada, 502
 - de una clase base virtual, 558
 - para adquirir un recurso, 643
 - por omisión, 358, 360
 - virtual, 528
- constructores de clases derivadas, 500
- contador, 179
- contenedores, 786, 810
- continue, 150
- control de acceso, 490
- conversión
 - con constructor, 446
 - de punteros y de referencias, 519
 - de tipos, ambigüedades, 451
 - entre tipos, 53
 - explícita o forzada, 55
- conversiones, 444
 - ascendentes, 562
 - cruzadas, 562
 - descendentes, 562
 - desde una clase virtual, 563
 - explícitas en una jerarquía de clases, 520
 - implícitas en una jerarquía de clases, 517
- copia de objetos, 503
- copiar una matriz, 237
- copy, 187, 253, 810
- copy_backward, 810
- cos, 850
- cosh, 850
- count, 810
- cout, 8, 99
- CPila, 740
- CR, 115
- CR+LF, 672
- creación de un programa, 6
- crear una nueva excepción, 633
- ctime, 854

D

- data, 187
- dato miembro, 337
- dec, 109
- decimal, 34
- declaración, 65
 - compleja, 264
 - de una función, 67
 - de una variable, 12
- decremento, 44
- define, 64
- definición de una función, 70
- definir una clase derivada, 494
- delegación, 382
- delete, 251, 365, 458
- depurador, 10
- depurar, 872
- deque, 786

derivación de plantillas, 605
destructor, 357, 364, 373
 de una clase derivada, 506
 para liberar un recurso, 646
 virtual, 531
devolver un puntero, 386
devolver una referencia, 387
devolver una referencia a un objeto constante,
 356
diccionario, 180
dirección de, 49
dirección de memoria, 75, 225
directrices, 63
directriz de inclusión, 64
 condicional, 65
directriz de sustitución, 64
directriz using, 9, 90
do ... while, 142
double, 15
dynamic_cast, 55, 521, 532, 563

E

E/S utilizando registros, 685
eliminar un elemento de una matriz, 392
else, 20
else if, 132
empty, 175, 190
end, 174
endl, 11, 109
ends, 109
enlazador, 9
entorno de desarrollo integrado, 863
enumeración, 30
eof, 104
EOF, 677
eofbit, 139, 199
equal, 810
erase, 257
errores en operaciones de E/S, 858
espacio de nombres, 9, 89
espacios en blanco, 26
especialización, 586, 587, 600
 explícita, 601
 parcial, 603
especificación de excepciones, 635
estado de un flujo, 104, 674
estructura
 como argumento, 290
 de un programa C++, 60
 else if, 132
 interna, 337

estructuras, 202
 abstractas de datos, 710
 acceso a sus miembros, 203
 crear, 202
 definir variables, 202
 dinámicas, 709
 miembros, 203
 operaciones, 204
 variables, 213
excepciones, 152, 623
 bad_alloc, 250
 bad_exception, 636
 capturar cualquier, 632
 capturar, 630
 como clase interna, 640
 crear, 633
 derivadas, 631
 especificación, 635
 flujo de ejecución, 637
 lanzar, 629
 manejar, 628
 no esperadas, 635
 relanzar, 633
 utilizar, 642
exception, 626
exceptions, 626
exit, 250
exp, 851
explicit, 372, 446
export, 592
expresión, 18
 condicional, 20
 booleana, 41
extern, 83

F

fabs, 852
fail, 104
failure, 654
false, 20
fclose, 858
fecha actual del sistema, 378
fecha/hora, 349
feof, 859
ferror, 858
fflush, 862
fgetc, 860
fgets, 861
fichero, 662
 de cabecera, 7
 ejecutable, 10

fichero
 fuente, 6
 objeto, 10
 binario, 667
 de cabecera, 833
 clases, 341
 de texto, 667
filebuf, 671
fill, 251, 810
fill_n, 810
fin de fichero, 114, 859
final de línea, 11
find, 175, 189, 810
find_end, 810
fixed, 109
flags, 108
float, 14
floatfield, 108
floor, 852
flujo, 98, 663
 como una condición, 103
 estado asociado, 104
 errores, 674
 excepciones, 674
flush, 109
 stream, 674
fopen, 857
for, 145
for_each, 201, 810
formato, 107
formato, especificaciones, 839
fprintf, 861
fputc, 860
fputs, 860
fread, 862
free, 857
freopen, 858
friend, 397, 515
front, 174
fscanf, 861
fseek, 860
fstream, 683
 constructor, 683
ftell, 859, 860
fugas de memoria, 252
función, 67
 copy, 253
 exit, 250
 fill, 251
 genérica, 582
 declaración, 582
 definición, 582

función
 localtime, 855
 memcpy, 253
 memset, 251
 miembro, 338
 pow, 309
 recursiva, 305
 strcmp, 303
funciones, 22
 en línea, 309
 sobrecargadas, 311
fwrite, 861

G

GCC, 4, 879
gcount, istream, 677
gdb, 6, 876
get, 112
get, istream, 676
getchar, 843
getline, 117, 190
 de string, 190, 199
 istream, 676
gets, 843
GNU, 4, 879
good, 104
goto, 150
GPL, 4

H

herencia, 483, 484
 múltiple, 490, 553
 ambigüedades, 555
 simple, 490
 virtual, 556
hermanas, clases, 553
hex, 109
hexadecimal, 34

I

identificadores, 36
if, 19, 127
 anidados, 129
ifstream, 681
 constructor, 681
ignore, 106
 istream, 676
implementación de una clase, 345
impresora, 687

include, 64
 inclusión, 380
 incremento, 44
 indirección, 49, 244
 iniciadores, lista, 361
 iniciar los elementos de un vector, 173
 inline, 309
 inorden, 759
 inserción, 801
 insert, 175, 188, 257, 267
 insertar nodo, 767
 insertar un elemento en una lista, 714
 instalación de CodeBlocks, 881
 instalación de Microsoft C/C++, 882
 int, 14
 integridad de los datos, 384
 interfaz, 338
 internal, 109
 invalid_argument, 625
 ios, 98
 bad(), 105
 badbit, 105
 eof(), 104
 eofbit, 105
 fail(), 104
 failbit, 105
 good(), 104
 goodbit, 105
 ios_base, 107
 failure, 625
 iostream, 98, 678
 is_open, filebuf, 673
 is_open, fstream, 684
 is_open, ifstream, 682
 is_open, ofstream, 680
 istream, 98, 675
 iteradores, 174

J

jerarquía de clases, 483, 490, 507

L

labs, 851
 lagunas de memoria, 252
 lanzar una excepción, 629
 leer caracteres, 112
 leer datos, 655, 692
 left, 109
 length, 190
 lenguaje máquina, 3

LF, 115
 lib, 6
 liberar objetos, ¿dónde?, 541, 566
 ligadura dinámica, 527
 ligadura estática, 527
 línea de órdenes, 301, 874
 Linux, 875, 877
 list, 786
 lista circular, 733
 doblemente enlazada, 746
 lista de iniciadores, 361
 clases derivadas, 500
 lista doblemente enlazada, 746
 lista lineal
 ordenada, 789
 simplemente enlazada, 710
 recorrer, 718
 literal de cadena de caracteres, 36
 literal de un solo carácter, 35
 literal entero, 34
 literal real, 35
 llamada a una función, 72
 local, objeto, 432
 localtime, 349, 854, 855
 log, 851
 log10, 851
 long, 14
 lower_bound, 810
 LPT1, 687

M

macros, 310
 main, 72
 malloc, 857
 manipuladores, 108
 map, 180, 786
 atributos y métodos, 182
 marca de fin de fichero, 138
 matriz, 166
 de objetos, 387
 de punteros, 241
 dentada, 194
 dinámica, 254
 asociativa, 178, 180
 de cadenas de caracteres, 197
 de estructuras, 206
 de matrices, 193
 de objetos string, 198
 de punteros a cadenas de caracteres, 246
 multidimensional, 191
 numérica multidimensional, 191

matriz
 static, 169
 acceder a un elemento, 168
 definir, 167
 iniciar, 169
 tipo y tamaño, 172
 max, 106, 655, 810
 max_size, 257
 máximo común divisor, 431
 memcmp, 856
 memcpy, 253, 856
 memset, 251, 855
 menús, 657
 merge, 810
 método
 abreviado, 350
 clear, 106
 constante, 354
 de inserción, 801
 de la burbuja, 798
 de quicksort, 802
 flags, 108
 get, 112
 heredado, redefinir, 513
 ignore, 106
 max, 106
 read, 686
 setf, 107
 sobrecargado, 348
 static, 378
 virtual, 523
 de una clase base virtual, redefinir, 560
 llamada, 524
 width, 107
 write, 686
 métodos, 338
 de una clase derivada, 496
 ocultos, 495
 por omisión, 495
 virtuales, implementación, 526
 virtuales, tabla, 527
 redefinir, 498
 miembros heredados, 491
 miembros que son punteros, 366
 min, 810
 MinGW, 879
 modificadores de acceso, 340
 modo de abrir un fichero, 671
 módulo, 59
 mutable, 355

N

new, 249, 364, 458
 nivel de indirección, 244
 noboolalpha, 108
 nodo de un árbol, 758
 nombre global, 81
 nombre local, 81
 noshowbase, 109
 noshowpoint, 109
 noshowpos, 109
 noskipws, 109
 not, 42
 not_eq, 41
 nothrow, 249
 numeric_limits, 28, 106, 655

O

objeto cin, 101
 objeto const, 354
 objeto cout, 99
 objeto de una clase derivada, construcción, 501
 objeto local, 432
 objeto miembro de una clase, 380
 objeto temporal, 433
 objetos estáticos frente a dinámicos, 540
 oct, 109
 octal, 34
 ocultación de datos, 339
 ofstream, 679
 constructor, 679
 open
 filebuf, 671
 fstream, 684
 ifstream, 681
 ofstream, 679
 operaciones con punteros, 228
 operador, 456
 -, 444
 delete, 461
 new, 459
 !, 442
 &, 49
 (), 454
 *, 49
 ::, 47
 [], 173, 453
 +, 427
 +=, 431
 <<, 99, 438
 =, 362, 451

- operador
 - =, 436
 - >>, 101, 441
 - coma, 48
 - condicional, 46
 - de ámbito, 497
 - de asignación, 451
 - de asignación por omisión, 504
 - de asignación, redefinir, 550
 - de conversión, 447
 - de extracción, 101
 - de extracción, istream, 677
 - de indexación, 453
 - de inserción, 99
 - de inserción, ostream, 675
 - delete, 251
 - desreferencia, 456
 - dynamic_cast, 532
 - llamada a función, 454
 - new, 249
 - sizeof, 48
 - sobrecargado, 421
 - ternario, 46
 - typeid, 535
 - operadores, 39
 - a nivel de bits, 43
 - aritméticos, 18, 39
 - con cadenas de caracteres, 472
 - de asignación, 43
 - de comparación, 20
 - de relación, 20, 40
 - lógicos, 41
 - prioridad, 52
 - sobrecargados, 314
 - unitarios, 42
 - operator, 422
 - or, 42
 - or_eq, 44
 - ordenación, 797
 - ordenar alfabéticamente una matriz de cadenas, 223
 - ordenar un fichero utilizando acceso aleatorio, 812
 - ostream, 98, 673
 - out_of_range, 625
 - overflow, método, 671
 - overflow_error, 625
- P**
- palabras clave, 36
 - parámetros, 68
 - actuales, 72
 - con valores por omisión, 350
 - de una plantilla, 580
 - formales, 71
 - por omisión, 307
 - por referencia, 233
 - path, 874
 - peek, 442
 - istream, 677
 - pila, 739
 - plantilla
 - de clase, 594
 - de clase, declaración previa, 599
 - de función, 582
 - definición, 580
 - list, 786
 - plantillas, 579, 722
 - de función, sobrecarga, 588
 - organizar código fuente, 590
 - valores por omisión, 608
 - polimorfismo, 536
 - polinomios, 265
 - pop_back, 175, 257
 - postorden, 759
 - pow, 309, 852
 - preorden, 759
 - preprocesador, 7, 63
 - printf, 838
 - prioridad de los operadores, 52
 - private, 341
 - programa, 3, 59
 - compuesto por varios ficheros, 78
 - ejecutable, 9
 - programación estructurada, 59
 - programación genérica, 579
 - programación orientada a objetos, 60
 - protected, 341
 - prototipo de una función, 68
 - public, 340
 - puntero, 75, 225
 - a un miembro, 401
 - a un puntero, 243
 - a una cadena de caracteres, 239
 - a una función, 315
 - como argumento, 286
 - como parámetro, 260
 - constante, 232
 - de escritura, 670
 - de lectura, 670
 - genérico, 231
 - inteligente, 649

puntero
 definir, 225
 punteros
 a estructuras, 258
 asignación, 229
 comparación, 231
 operaciones aritméticas, 229
 operadores, 227
 y matrices, 235
 y referencias a clase derivadas, 516
 push_back, 175, 255, 257
 put, ostream, 674
 putback, istream, 677
 putchar, 843
 puts, 844

Q

queue, 786
 quicksort, 802

R

raíz cuadrada, 19
 raíz de un árbol, 758
 rand, 852
 rbegin, 174
 rdbuf, fstream, 684
 rdbuf, ifstream, 682
 read, 686
 istream, 676
 realloc, 857
 reasignar un bloque de memoria, 252
 recorrer un árbol, 759
 redefinición de un método virtual, 530
 redefinir control de acceso, 499
 redefinir métodos de la clase base, 498
 redefinir un método heredado, 513
 redefinir un método virtual, 523
 redimensionar una matriz, 257
 redireccionar la entrada o salida, 303
 referencia, 50, 77, 233
 valor retornado como, 299
 register, 83
 registro, 662, 685
 reinterpret_cast, 55
 remove, 810
 rend, 174
 replace, 189, 810
 reservar de memoria, 258, 392
 reserve, 257
 resize, 175, 190, 255, 257

nulo, 232

retornar la dirección de una v. static, 297
 retornar un puntero, 295
 retornar una copia de los datos, 293
 return, 70, 250
 reverse, 810
 rewind, 860
 right, 109
 RTTI, 535
 runtime_error, 626

S

salida con formato, 107
 scanf, 841
 scientific, 109
 search, 810
 search_n, 810
 secuencia de escape, 27
 seekg, istream, 677, 698
 seekp, ostream, 675, 699
 sentencia
 break, 149
 compuesta, 66
 continue, 150
 de asignación, 15
 do ... while, 142
 for, 145
 for_each, 201
 goto, 150
 if, 19, 127
 return, 70
 simple, 66
 switch, 134
 while, 137
 try ... catch, 152
 set, 786, 874
 set_unexpected, 637
 setbase, 109
 setf, 107, 109
 setfill, 109
 setiosflags, 109
 setprecision, 109
 setw, 109
 short, 14
 showbase, 109
 showpoint, 109
 showpos, 109
 signed, 29
 sin, 850
 sinh, 850
 sinónimos de otro tipo, 33
 size, 175, 190, 257

sizeof, 48
 skipws, 109
 sobrecarga, 99

- de funciones, 311
- de los operadores ++ y --, 442
- de los operadores unarios/binarios, 444
- de métodos, 349
- de operadores binarios, 428
- de un operador, propiedades, 424
- de un operador, restricciones, 428
- del operador !, 442
- del operador <<, 437
- del operador =, 362
- del operador ==, 436
- del operador >>, 440
- del operador delete, 461
- del operador new, 459
 - resolución, 589

 software libre, 4
 sort, 810
 sprintf, 848
 sqrt, 19, 852
 srand, 853
 stack, 786
 static, 83, 375
 static struct tm, 349
 static_cast, 55, 520, 563
 std, 9, 89
 STL, 833
 strcat, 844
 strchr, 844
 strcmp, 303, 845
 strepy, 844
 strcspn, 845
 stream, 663
 streambuf, 670
 string, 15, 117, 185, 199, 254, 610

- acceso a un carácter, 186
- asignación, 186
- búsqueda, 189
- comparaciones, 187
- concatenación, 189
- constructor, 185
- convertir, 187
- E/S, 190
- inserción, 188
- reemplazar, 189
- subcadenas, 189
- tamaño, 190

 strlen, 845
 strlwr, 188, 847
 strncat, 845

strncmp, 846
 strncpy, 846
 strrchr, 844
 strspn, 846
 strstr, 846
 strtok, 847
 struct, 202
 strupr, 188, 847
 subclase, 483
 subíndice, 167
 substr, 190
 superclase, 483
 swap, 810
 switch, 134
 sync_with_stdio, 837
 system, 862

T

tabla de métodos virtuales, 527
 tabla-v, 527
 tan, 850
 tanh, 851
 tellg, clase istream, 677, 699
 tellp, ostream, 675, 699
 template, 580
 terminate, 635
 this, 352
 throw, 152, 629, 635
 time, 349, 853
 tipo, 840

- abstracto de datos, 424
- bool, 29
- char, 14
- double, 15
- float, 14
- int, 14
- long, 14
- polimórfico, 536
- short, 14
- string, 15, 117
- unsigned int, 14
- unsigned long, 14
- unsigned short, 14
- wchar_t, 29

 tipos de datos, 28
 tipos de valores, 13
 tipos derivados, 29
 tipos primitivos, 28
 toascii, 849
 tolower, 849
 toupper, 849

transform, 810
true, 20
try, 152, 630
typedef, 33, 172
typeid, 535
typename, 581

U

underflow, método, 671
unexpected, 635
unget, istream, 677
unión, 208
unique, 810
unsetf, 109
unsigned, 29
unsigned int, 14
unsigned long, 14
unsigned short, 14
uppercase, 109
using, 9, 90

V

valor retornado por una función, 22
valor-i, 424

variable, 12, 38
 iniciar, 39
variables locales, 13
vector, 172, 254, 786
 insertar por el principio, 267
virtual, 523
 clase base, 556
Visual Studio .NET, 867
void, 22, 69, 70
void *, 231

W

wchar_t, 29, 35
what, 626
while, 137
while, do, o for anidados, 140
width, 107
write, 686
 ostream, 674
ws, 109

X

xor, 43
xor_eq, 44

Del mismo autor

- Curso de programación con **PASCAL** ISBN: 978-84-86381-36-3
224 págs.
 - Curso de programación **GW BASIC/BASICA** ISBN: 978-84-86381-87-5
320 págs.
 - Manual para **TURBO BASIC** ISBN: 978-84-86381-43-1
Guía del programador 444 págs.
 - Manual para **Quick C 2** ISBN: 978-84-86381-65-3
Guía del programador 540 págs.
 - Manual para **Quick BASIC 4.5** ISBN: 978-84-86381-74-5
Guía del programador 496 págs.
 - Curso de programación **Microsoft COBOL** ISBN: 978-84-7897-001-8
480 págs.
 - Enciclopedia del lenguaje **C** ISBN: 978-84-7897-053-7
888 págs.
 - Curso de programación **QBASIC y MS-DOS 5** ISBN: 978-84-7897-059-9
384 págs.
 - Curso de programación **RM/COBOL-85** ISBN: 978-84-7897-070-4
396 págs.
 - El abecé de **MS-DOS 6** ISBN: 978-84-7897-114-5
224 págs.
 - Microsoft **Visual C ++** (ver. 1.5x de 16 bits) ISBN: 978-84-7897-180-0
Aplicaciones para Windows 846 págs. + 2 disquetes
 - Microsoft **Visual C ++** ISBN: 978-84-7897-561-7
Aplicaciones para Win32 (2ª edición) 792 págs. + disquete
 - Microsoft **Visual C ++** ISBN: 978-84-7897-344-6
Programación avanzada en Win32 888 págs. + CD-ROM
 - **Visual Basic 6** ISBN: 978-84-7897-357-6
Curso de programación (2ª edición) 528 págs. + disquete
 - Enciclopedia de Microsoft **Visual Basic 6** ISBN: 978-84-7897-386-6
1072 págs. + CD-ROM
 - El lenguaje de programación **Java** ISBN: 978-84-7897-485-6
320 págs. + CD-ROM
 - El lenguaje de programación **C#** ISBN: 978-84-7897-500-6
320 págs. + CD-ROM
-

Del mismo autor

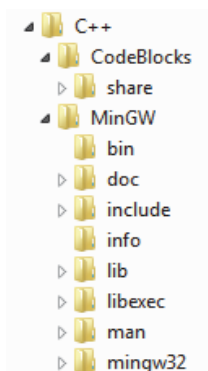
- El lenguaje de programación **Visual Basic.NET** ISBN: 978-84-7897-525-9
464 págs. + CD-ROM
 - Enciclopedia del lenguaje **C ++** ISBN: 978-84-7897-584-6
1120 págs. + CD-ROM
 - **Java 2** ISBN: 978-84-7897-686-7
Curso de programación (3ª edición) 880 págs. + CD-ROM
 - **Java 2**. Interfaces gráficas y Aplicaciones para Internet (2ª edición) ISBN: 978-84-7897-692-8
592 págs. + CD-ROM
 - Enciclopedia de Microsoft **Visual Basic** ISBN: 978-84-7897-710-9
976 págs. + CD-ROM
 - **Microsoft C#** ISBN: 978-84-7897-737-6
Curso de programación 866 págs. + CD-ROM
 - **Microsoft Visual Basic .NET** ISBN: 978-84-7897-740-6
Lenguaje y aplicaciones (2ª edición) 520 págs. + CD-ROM
 - **Java 2** ISBN: 978-84-7897-745-1
Lenguaje y aplicaciones 392 págs. + CD-ROM
 - Programación orientada a objetos con **C ++** (4ª edición) ISBN: 978-84-7897-761-1
648 págs. + CD-ROM
 - **C/C++** ISBN: 978-84-7897-762-8
Curso de programación (3ª edición) 708 págs. + CD-ROM
 - Enciclopedia de Microsoft **Visual C#** (2ª edición) ISBN: 978-84-7897-810-6
1012 págs. + CD-ROM
 - **Microsoft Visual Basic .NET** ISBN: 978-84-7897-812-0
Curso de programación 832 págs. + CD-ROM
 - **Microsoft C#** ISBN: 978-84-7897-813-7
Lenguaje y aplicaciones (2ª edición) 520 págs. + CD-ROM
 - **Java 2**. Interfaces gráficas y aplicaciones para Internet (3ª edición) ISBN: 978-84-7897-859-5
718 págs. + CD-ROM
 - **Aplicaciones .Net multiplataforma** (Proyecto Mono) ISBN: 978-84-7897-880-9
212 págs. + CD-ROM
-

INSTALACIÓN

Para instalar el kit de desarrollo de C++ y los ejemplos de este libro siga los pasos indicados a continuación:

PLATAFORMA WINDOWS

La instalación del entorno integrado *CodeBlocks* y del compilador *MinGW* puede realizarse de un solo paso si ejecuta el fichero *codeblocks-8.02mingw-setup.exe* localizado en la carpeta EDI del CD. También puede realizarla instalando por separado el EDI y MinGW según muestra la figura siguiente (véase el apéndice E):



Una vez instalado el entorno de desarrollo integrado (EDI), puede personalizarlo a través de las órdenes *Editor...* y *Environment...* del menú *Settings*.

Para la instalación del compilador C/C++ de Microsoft, vea el apéndice E.

EJEMPLOS DEL LIBRO:

Los ejemplos del libro puede instalarlos en una carpeta de su gusto o los puede recuperar directamente desde el CD cuando los quiera consultar. La forma de descargar el CD se indica en el prólogo.

PLATAFORMA LINUX:

Las distribuciones GNU/Linux proporcionan implementaciones GCC.

LICENCIA

Todo el contenido de este CD, excepto los ejemplos del libro, es propiedad de las firmas que los representan. La inclusión en este libro se debe a su gentileza y es totalmente gratuita y con la finalidad de apoyar el aprendizaje del software correspondiente. Para obtener más información y actualizaciones visite las direcciones indicadas en dicho software.

Al realizar el proceso de instalación, haga el favor de consultar el acuerdo de licencia para cada uno de los productos.

WEB DEL AUTOR: <http://www.fjceballos.es>

En esta Web podrá echar una ojeada a mis publicaciones más recientes y acceder a la descarga del software necesario para el estudio de esta obra así como a otros recursos.

Enciclopedia del lenguaje

C++

La programación orientada a objetos (POO) es una de las técnicas más modernas de desarrollo que trata de disminuir el coste del software, aumentando la eficiencia y reduciendo el tiempo de espera. Por eso, donde la POO toma verdadera ventaja es en el poder com-partir y reutilizar el código.

Sin embargo, no debe pensarse que esta forma de programación resuelve todos los problemas de desarrollo de una forma sencilla y rápida. Para conseguir buenos resultados, es preciso dedicar un tiempo mayor al análisis y al diseño, pero no será un tiempo perdido, ya que redundará en el empleado en la realización de aplicaciones futuras.

Existen varios lenguajes que permiten escribir un programa orientado a objetos; entre ellos hemos elegido C++. ¿Por qué C++? Porque posee características superiores a otros lenguajes. Las más importantes son programación modular y orientada a objetos, portabilidad, brevedad, compatibilidad con C y velocidad.

Además, se trata de un lenguaje de programación estandarizado (ISO/IEC), ampliamente difundido, y con una biblioteca estándar C++ que lo ha convertido en un lenguaje universal, de propósito general, y ampliamente utilizado tanto en el ámbito profesional como en el educativo.

Enciclopedia del lenguaje C++ es un libro:

- Válido para plataformas Windows y Unix/Linux.
- Totalmente actualizado al estándar ISO/IEC, relativo al lenguaje C++ estándar.
- Con ejemplos claros y sencillos, fáciles de entender, que ilustran los fundamentos de la programación C++.
- Que le permitirá aprender lógica de programación.
- Que le permitirá aprender programación orientada a objetos.
- Que le enseñará a trabajar con estructuras estáticas y dinámicas de datos, ficheros y excepciones.
- Con el que aprenderá a utilizar las plantillas más comunes de la biblioteca estándar de C++ (STL).
- Con el que adquirirá unos elevados conocimientos en la POO.
- Y con el que le será fácil aprender a desarrollar aplicaciones.

ra-ma.es

Podrá descargarse desde www.ra-ma.es, en la página Web correspondiente al libro, un CD-ROM con todos los ejemplos realizados y con las direcciones de Internet desde las que se podrá descargar el software necesario para que el lector pueda reproducirlos durante el estudio.



Ra-Ma®