Fundamentos de C++

Segunda edición

1.ª reimpresión

Gerardo Aburruzaga García • Inmaculada Medina Bulo
Francisco Palomo Lozano

Profesores del Departamento de Lenguajes y Sistemas Informáticos
Escuela Superior de Ingeniería • Universidad de Cádiz

Cádiz, 2009

Fundamentos de C++

Segunda edición

1.ª reimpresión

Gerardo Aburruzaga García
`Gerardo.Aburruzaga@uca.es`

Inmaculada Medina Bulo
`Inmaculada.Medina@uca.es`

Francisco Palomo Lozano
`Francisco.Palomo@uca.es`

Edita: Servicio de Publicaciones de la Universidad de Cádiz
C/ Doctor Marañón, 3. 11002 Cádiz (España)
`http://www.uca.es/publicaciones`

Agradecimientos

Gerardo A la Pati, la niña más bonita del mundo (o eso creo yo, pero si la vieran quizá no me negarían la razón). A su madre, que no sólo dio a luz a ella sino a mí. A mis amigos Inma y Paco, que me permiten seguir siendo su compañero, o viceversa. A mis padres, por razones obvias.

Inma A Paquito y Alejandrito, mis tesoros.

A mis hermanos, Mª Ángeles, Mª Carmen, Miguel y Fernando, con cariño y admiración. Nunca olvidaré los buenos ratos de diversión pasados juntos.

A Paco y Gerardo, respectivamente, magnífico marido y gran amigo, además de compañeros y co-autores. Agradezco sinceramente vuestra paciencia y cooperación mientras añadíamos, cambiábamos de opinión, eliminábamos, recortábamos y corregíamos.

Paco A Paquito y Alejandrito, tesoros de Dios.

A mi familia, y especialmente a mi abuelo Francisco, que desde el Cielo me sigue ayudando. Simplemente, ellos lo merecen.

A Inma, por su apoyo y dedicación durante todos estos años.

A Gerardo, juntos comenzamos nuestra andadura por este maravilloso lenguaje. Él me enseñó mucho de lo que sé.

Además, estamos profundamente agradecidos a Francisco Periáñez Gómez por sus minuciosas correcciones y su cooperación en todo momento.

También queremos expresar nuestro agradecimiento a nuestros alumnos de la Escuela Superior de Ingeniería de Cádiz. Ellos han sido el principal motivo por el que hemos escrito este libro.

Y por último, no podemos olvidarnos de alguien muy importante; gracias Bjarne por haber creado este gran lenguaje que tantas horas de entretenimiento nos ha proporcionado.

Índice general

Índice de cuadros

Índice de figuras

Prefacio

Este libro explica el lenguaje de programación C++ según la norma aprobada hace unos pocos años por el esfuerzo conjunto de los organismos de normalización ANSI e ISO. Lo hemos escrito pensando en nuestros alumnos de Programación Orientada a Objetos, asignatura de segundo curso de las Ingenierías Técnicas en Informática en la Universidad de Cádiz. Ellos ya han visto el lenguaje C en primer curso y por tanto sólo necesitan un breve repaso. Esto nos viene bien porque C++ es un lenguaje con muchas características y si ya se conocen las básicas incluidas en C se tiene algo ganado; por lo tanto nuestro libro parte de la base de que ya se poseen conocimientos de C, aunque también es posible utilizarlo sin tenerlos puesto que de todas formas no se omite la parte básica de C++, es decir la parte relativa a la programación estructurada que comparte con su antecesor.

A pesar de utilizarse en clases de prácticas de dos horas, el libro no está dividido en «prácticas» sino en capítulos extensos, y no hay que pensar que cada capítulo pueda estudiarse en dos horas. Si se usa para clases prácticas, es criterio del profesor hasta dónde llegar en cada capítulo, dependiendo de sus alumnos y de otras circunstancias. Nosotros estimamos que unas quince prácticas de dos horas cada una es el *mínimo* suficiente.

Aunque C++ admita varios paradigmas de programación, el empleo principal de este lenguaje es en la programación orientada a objetos (P.O.O.); se recomienda también por tanto para aprovechar al máximo la lectura, el conocimiento previo de alguna metodología orientada a objetos y de P.O.O. en general; si bien no es imprescindible este conocimiento pues se dan nociones generales básicas sobre la P.O.O.

El libro se divide en siete capítulos. A continuación se detallan sus contenidos.

1. Se empieza explicando cuándo, cómo y por qué surgió este lenguaje, lo que entronca con un breve resumen del paradigma de la P.O.O. A continuación, para pasar ya rápidamente a la práctica, se habla del proceso de la compilación de un programa; esta parte está ligeramente sesgada a entornos Unix no gráficos porque pensamos que si no es la

forma más cómoda sí es la más didáctica, pues se «ve» lo que ocurre. No podía faltar el primer programa, el «programa que saluda», típico de cualquier curso de cualquier lenguaje de programación; se aprovecha este programa para analizar las diferencias con el correspondiente escrito en C y se siguen analizando otras diferencias importantes.

Por último se hace un breve recorrido por casi todo el lenguaje, salvo la herencia. No se pretende salir sabiendo ya C++ sino tener una visión general de lo que se va a ver en capítulos sucesivos. No se debe preocupar el lector mucho, por lo tanto, si no entiende completamente lo que aquí se muestra porque se verá todo más adelante, salvo los espacios de nombres.

2. Este capítulo y el siguiente tratan de los aspectos básicos del C++, lo que tiene en común con C y con otros lenguajes de programación estructurada. En éste se enumeran los tipos básicos de datos y los operadores incorporados; tanto los que comparte con C como los nuevos. Además se introducen dos tipos definidos en la biblioteca estándar: *string* y *vector*, que por su utilidad se emplearán muy pronto.

3. Aquí se estudian brevemente las estructuras de control, que son las mismas que en C, por lo que este capítulo también sirve de repaso; se incluyen las funciones, con las características nuevas de expansión en línea, parámetros por omisión, sobrecarga, etc., y se recalcan siempre las diferencias que pueda haber con C.

4. Éste quizá es el capítulo más denso y que merece un mayor estudio y esfuerzo por parte del lector, pues habla de la característica clave de C++: las clases y los objetos. Incluye desde la definición básica de tipos concretos definidos por el usuario, hasta la herencia y el polimorfismo, bases de la P.O.O. Se incluye un apartado algo extenso sobre las relaciones entre clases, básico para entender cómo se construye un programa completo en C++. Aunque es un tema avanzado, no se omite la herencia múltiple ni la identificación de tipos en tiempo de ejecución (RTTI).

   Por último se habla de la programación genérica mediante las plantillas, base de la construcción de la biblioteca estándar de C++ y de otras muchas donde la eficiencia (uno de los objetivos del lenguaje) sea importante.

5. En este capítulo se trata del mecanismo de excepciones, que no puede faltar en una biblioteca bien diseñada ni en un programa de mediana complejidad. Este mecanismo permite dividir los errores o «excepciones» en dos partes: la detección y el tratamiento.

6. Una vez visto el lenguaje en su totalidad, se estudia la parte más importante de la biblioteca estándar en este capítulo y el siguiente. Primero, por su extensión e importancia, la parte dedicada a entrada y salida (E/S), los flujos.

7. Y por último, la biblioteca estándar de plantillas (STL), dedicada a clases contenedoras, que proporciona una base sólida para emplear estructuras básicas de datos y algoritmos generales sobre ellas. Se ha omitido la parte de la biblioteca estándar dedicada al soporte del lenguaje (aunque algo se ve donde es necesario referirse a ella) y la dedicada a cálculo numérico; se supone que ya se conoce la biblioteca estándar de C, incluida en la de C++ y de la que se da bibliografía.

Los autores

Convenios tipográficos

Al escribir este libro se ha empleado un conjunto de convenios tipográficos para marcar órdenes, ficheros, funciones, etc. Se detallan a continuación.

`while` Una palabra reservada del lenguaje C++.

ordenar() Una función, de la biblioteca o no. Esto no significa que no reciba ningún parámetro ni que no devuelva nada; eso se indica en el prototipo. Los paréntesis simplemente significan que se trata del identificador de una función.

NULL Una macro sin parámetros, o constante simbólica.

assert() Una macro con parámetros, o pseudo-función. Aplíquese lo dicho para ellas.

double El nombre de un tipo definido por el lenguaje o por el programador.

`datos.dat` Un nombre o camino de fichero.

`<iostream>` Un fichero de cabecera estándar.

`"dni.h"` Un fichero de cabecera definido por el programador.

`c++` Una orden o programa ejecutable.

n Un parámetro formal de una macro o función, o una variable.

`Tipo de máquina de escribir` Empleado para programas, funciones o trozos de código de ejemplos; también para órdenes y para prototipos de funciones o macros, salvo los parámetros.

Capítulo 1

Introducción al lenguaje C++

1.1. Historia

Fue a finales de la década de los 60 cuando se propuso por primera vez una nueva forma de programación[1] que permitía el desarrollo de *software* de forma estructurada y modular, ofreciendo al mismo tiempo grandes posibilidades de *abstracción* y por lo tanto de *reutilización*.

Uno de los primeros lenguajes con el distintivo de «orientado a objetos» fue SmallTalk, al que inicialmente no se le prestó demasiada atención en el mundo de la industria. Uno de los factores que intervino fue, quizás, cierta ineficiencia intrínseca a las primeras implementaciones del lenguaje (el lenguaje era *interpretado*), lo que unido a la menor potencia de cálculo existente en aquel entonces lo hacía relativamente lento.

La orientación a objetos permaneció latente dentro del mundo académico y no fue sino hasta mediados de la década de los 80 cuando volvió a resurgir con fuerza y aparecieron nuevos lenguajes orientados a objetos, mucho más desarrollados, que rápidamente entraron en el mundo de la industria como es el caso de C++.

C++ fue inventado por Bjarne Stroustrup (fig. 1.1), mientras trabajaba para los laboratorios Bell de AT&T, en el año 1985 aproximadamente. Su nombre deriva del operador incremento del lenguaje C; se podría decir que C++ es un «C incrementado», un paso más en el C[2].

El lenguaje C es un lenguaje estructurado y fue inventado por Dennis M.

[1] La *programación orientada a objetos* o POO.

[2] Se bromeaba con la posibilidad de que el sucesor deC se llamara D, por ser la siguiente letra del alfabeto en la secuencia B, C; o P, por ser la siguiente letra de BCPL (*Basic Combined Programming Language*), un antecesor lejano de C.

(a) Dennis Ritchie (b) Bjarne Stroustrup

Figura 1.1: Los creadores de C y C++

Ritchie (fig. 1.1), que también trabajaba para los laboratorios Bell de AT&T, a principios de los años 70. Fue creado a partir del lenguaje B, el cual venía a su vez del BCPL, con el propósito de escribir el sistema operativo UNIX[3] en un lenguaje de más alto nivel que el ensamblador.

> El nombre UNIX es en realidad una deformación de UNICS, que era una simplificación de un proyecto anterior de sistema operativo, llamado MULTICS, con el que se pretendía controlar una ciudad entera, y que fue abandonado tras una enorme inversión.

Con la distribución gratuita en las universidades del UNIX junto con un compilador de C y el código fuente, vino el éxito del sistema y del lenguaje, y con este éxito también la necesidad de adaptarlo a más propósitos de aquél para el que fue concebido.

A lo largo de los años han ido surgiendo muchos dialectos o variantes de C y también nuevos lenguajes basados en él.

Con la constitución en 1985 del comité X3J11 de ANSI (*American National Standards Institute*) para crear un estándar del lenguaje, el problema de los dialectos empezó a resolverse, y desde 1989 ya lo está.

Un año después, ISO (*International Standards Organization*) creó su propio estándar de C equivalente al de ANSI, pero ligeramente ampliado para facilitar su empleo en países de habla no inglesa. En septiembre de 1994 se introdujo un nuevo anexo que incluía algunas ampliaciones menores (como

[3] UNIX® es una marca registrada de X/Open.

la aparición de la cabecera `<iso646.h>`, lo que se conoce como «la propuesta danesa»).

Por lo tanto, si alguien quiere que su programa en C sea entendido por cualquiera que sepa C, o incluso que se pueda transportar a máquinas distintas, debe emplear C ANSI/ISO.

No obstante, para resolver ciertos problemas de programación, C se queda corto; sin embargo, al ser tan popular y querido por los programadores, se han inventado nuevos lenguajes basados en él: C concurrente, C objetivo (Objective C), C paralelo, y sobre todos destaca C++.

C++ es un lenguaje orientado a objetos, aunque no «puro», debido a que soporta otros estilos de programación como el estructurado. Por esto, también, se suele decir que es un *lenguaje híbrido* o que no es un *lenguaje orientado a objetos* puro.

En palabras de Bjarne Stroustrup, el hecho de que C++ no sea un lenguaje orientado a objetos puro es más una ventaja que un inconveniente, ya que lo hace más versátil y adecuado para un mayor número de aplicaciones.

Ante la fuerte expansión del lenguaje en el mundo académico y empresarial, y para evitar los problemas de diversificación que previamente había sufrido C, se reconoció rápidamente la necesidad de normalizarlo. Con este objetivo, se constituyó en 1989 el comité X3J16 de ANSI, que terminó su trabajo a finales de 1997.

Al igual que ocurre con C, si alguien quiere que su código sea transportable debe emplear una versión normalizada del lenguaje, es decir, C++ ANSI.

En sus orígenes, C++ incorporó conceptos que de un modo u otro habían ya existido en otros lenguajes distintos de C. Por ejemplo, el concepto de «clase», fundamental para la programación orientada a objetos, ya estaba presente en Simula67, un lenguaje orientado a la simulación de procesos. Tanto es así, que las primeras versiones de C++ que Bjarne Stroustrup creó, fueron extensiones de C denominadas «C con clases» y que al igual que Simula67 tenían como objetivo su empleo en simulación.

Como conclusión se puede decir que C++ ha evolucionado hasta convertirse en un lenguaje de programación de propósito general, ligeramente sesgado hacia la programación de sistemas, que:

- Constituye un avance respecto al lenguaje C, considerándose en este sentido como un «C mejorado».
- Facilita la abstracción de datos y operaciones, particularmente a través del paradigma de la orientación a objetos.

- Permite también trabajar con los paradigmas de la programación estructurada y de la programación genérica, e incluso mezclar los tres paradigmas.

1.2. El paradigma de la orientación a objetos

A continuación se realizará una breve introducción a los conceptos básicos de la orientación a objetos, presentando las ventajas que supone emplear un enfoque orientado a objetos en el *proceso de desarrollo de software*.

La idea fundamental tras este paradigma estriba en tratar de organizar el sistema en torno a los objetos que intervienen en él, en vez de hacerlo alrededor de los procesos y los datos, que es como se lleva a cabo en las *metodologías estructuradas*.

Vivimos en un mundo de objetos, por esto no es sorprendente que se hayan propuesto distintas *metodologías orientadas a objetos*: abstracciones que modelan el mundo real empleando sus propios elementos con el objetivo de ayudar a entenderlo mejor.

Entre las razones que hacen tan atractiva la orientación a objetos figuran:

- La relativa cercanía de sus conceptos a las entidades que aparecen en el mundo real.
- La simplicidad del modelo, que emplea los mismos elementos fundamentales para expresar de manera uniforme el análisis, el diseño y la implementación de un sistema.
- La gran capacidad de adaptación e integración de sus modelos, que facilita la realización de modificaciones, incluso durante el proceso de desarrollo, y el mantenimiento.
- La posibilidad de aumentar las oportunidades de reutilización de *componentes de software* en proyectos distintos.

1.2.1. Elementos fundamentales

Durante muchos años el término «orientado a objetos» se empleó como sinónimo de una cierta forma de programar que se empleaba con los lenguajes de programación orientados a objetos (principalmente SmallTalk) y que poseía características que la distinguían de la programación estructurada tradicional.

Hoy en día el paradigma de la orientación a objetos ofrece una visión más amplia e integrada del proceso de desarrollo de software[4] a través de diversas metodologías que abarcan el ciclo de vida completo de una aplicación.

En un *sistema orientado a objetos*, el software se organiza como un conjunto finito de objetos que contienen tanto datos como operaciones y que se comunican entre sí mediante mensajes.

Existe una serie de pasos que sirven de guía a la hora de modelar[5] un sistema empleando orientación a objetos:

1. Identificar los *objetos* que intervienen en él.
2. Agrupar en *clases* a todos aquellos objetos que tengan características y comportamiento comunes.
3. Identificar los *datos* y *operaciones* de cada una de las clases.
4. Identificar las *relaciones* que puedan existir entre las clases.

Por ejemplo, si se está modelando un sistema de autoedición en una editorial se pueden identificar a primera vista diversos objetos, como libros, índices, capítulos, figuras, tablas, y autores.

Esta información se obtiene de la observación directa del sistema. La editorial compone un libro a partir de una serie de capítulos individuales, tablas y figuras suministradas por los autores; el libro ha de completarse con un índice. También debe mantener en su base de datos información sobre los autores del libro: debe ser posible averiguar con facilidad los datos personales de cada autor, qué otros libros ha publicado con la editorial, etc.

Todos los libros tienen características comunes; igual ocurre con los restantes conjuntos de objetos. Esto lleva a agrupar a estos objetos en clases. El objeto es un elemento individual con su propia identidad, por ejemplo, el libro[6] *El Lenguaje de Programación C++* ; la clase es la colección de objetos similares, por ejemplo, el conjunto de todos los libros forma una clase.

Cada clase tiene sus propias características y comportamiento. Por ejemplo, de un autor se pueden guardar datos como su nombre, apellidos, dirección y teléfono. También sobre un autor se pueden realizar distintas operaciones, como imprimir sus datos o cambiar su número de teléfono.

Entre las clases pueden existir distintos tipos de relaciones. Por ejemplo, entre las clases *Autor* y *Libro* existe una relación de *asociación* obvia: cada

[4] Este es uno de los cometidos de la *Ingeniería del Software*.

[5] Nos referimos aquí a la obtención de un *modelo estático* que es el que refleja las características estructurales de un sistema.

[6] La identidad de un libro, que lo distingue de todos los demás, viene dada por su ISBN.

autor ha escrito uno o varios libros y, recíprocamente, cada libro tiene uno o varios autores.

También es cierto que todos los objetos de la clase *Libro* se componen de otros objetos de la clase *Capítulo*: existe, pues, lo que se denomina una relación de *agregación* entre ambas clases.

De forma similar, la clase *Autor* se puede definir a partir de una clase más general que hasta ahora no habíamos tenido en cuenta: la clase *Persona*, que describe los datos y operaciones habituales de las personas. Podemos considerar la clase *Persona* como una *generalización* de la clase *Autor* o, recíprocamente, que esta clase es una *especialización* de la clase *Persona*.

Como se observa, tanto las relaciones de agregación como las de generalización/especialización permiten apoyar la definición de una clase en las de otras.

A continuación ofrecemos una visión general de los principios en los que se sustenta la orientación a objetos. Un lenguaje orientado a objetos debe favorecer la utilización de todos ellos; tanto es así, que estos principios se han presentado como una caracterización de este tipo de lenguajes.

También hay que decir que no son exclusivos de la orientación a objetos. Por ejemplo, la *abstracción* es común a, prácticamente, todos los paradigmas, el *encapsulado* aparece en los lenguajes modulares, el *polimorfismo* en los funcionales, etc.

Abstracción

Abstraer consiste en considerar sólo aquellos aspectos de un problema que son importantes desde un cierto punto de vista y soslayar el resto. Evidentemente, aspectos superfluos en unos casos pueden ser de importancia capital en otros: todo depende del problema que se esté resolviendo.

En el modelado orientado a objetos de un sistema esto significa centrarse en *qué es* y *qué hace* un objeto antes de decidir *cómo* debe implementarse. La abstracción aumenta nuestra libertad a la hora de tomar decisiones evitando compromisos prematuros con los detalles concretos del sistema que se está modelando.

La abstracción posee diversos grados denominados *niveles de abstracción*. Los niveles de abstracción ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real.

Por ejemplo, cuando se realiza el análisis de un sistema sabemos que hay que concentrarse en qué hace ese sistema, pero no en cómo lo hace. A este nivel de abstracción resulta completamente irrelevante el hecho de que los

datos de una persona sean leídos de una tarjeta con código de barras o suministrados a través de un teclado. En cambio, desde el punto de vista del programador, situado en un nivel de abstracción muy inferior, estos aspectos sí son importantes.

Es durante este proceso de abstracción cuando se decide qué características y comportamiento se deben retener en el modelo. Un modelo en el que se retiene más de lo necesario puede dar lugar a redundancias y ambigüedades; un modelo en el que no se retiene lo suficiente puede carecer de utilidad o incumplir los requisitos exigidos.

La abstracción es el principio fundamental que se encuentra tras la *reutilización*. Tan sólo se puede reutilizar un componente si en él se ha abstraído la esencia de un conjunto de elementos del mundo real que aparecen una y otra vez, con ligeras variantes, en sistemas diferentes.

Encapsulado

Encapsular significa reunir, dotando de una cierta estructura, a todos los elementos que a un cierto nivel de abstracción se pueden considerar pertenecientes a una misma entidad.

El encapsulado es también el proceso de agrupar datos y operaciones relacionados bajo una misma unidad de programación. Esto permite aumentar la *cohesión* de los componentes del sistema.

Así, por ejemplo, los objetos que poseen las mismas características y comportamiento se agrupan en clases, que no son más que unidades de programación que encapsulan datos y operaciones.

Ocultación de datos

La ocultación de datos permite separar el aspecto de un componente, que viene determinado por su *interfaz* con el exterior, de sus detalles internos de implementación. La interfaz del componente representa un «contrato» de prestación de servicios entre él y los demás componentes del sistema.

Así los clientes de un componente sólo necesitan conocer los servicios que éste ofrece, no cómo están realizados internamente[7].

Esto permite disminuir las interdependencias entre los componentes relacionados del sistema, reduciendo el *acoplamiento* entre ellos.

[7] Algunos autores confunden los conceptos de encapsulado y ocultación de datos; creemos que esto se debe, principalmente, a que se suelen emplear conjuntamente. Esperamos que el lector comprenda que, aunque relacionados, son distintos.

Esto es importante, ya que a la hora de implementar un componente pueden aparecer detalles irrelevantes para su uso. Dichos detalles de implementación suelen surgir por la necesidad de utilizar estructuras de datos y funciones auxiliares.

No se debe permitir que estos detalles empañen la visión externa que sus clientes tienen de él. Por tanto debe existir un mecanismo que permita decidir qué es lo que puede o no «verse» desde el exterior.

Por ejemplo, de este modo se puede modificar la implementación de una clase sin afectar a las restantes relacionadas con ella. Sólo hay que mantener el contrato.

En general, esto implica que no se debe permitir que los datos de un objeto sean modificados directamente desde el exterior. Esto se suele conseguir especificando en la definición de su clase que éstos deben permanecer ocultos y proporcionando operaciones adicionales que permitan observar los datos de interés y realizar su modificación de manera controlada.

Generalización

Generalizar consiste en no resolver nunca un problema concreto sin antes pensar que, probablemente, sea un caso particular de un problema mucho más general. Si se encuentra una solución al problema general, se habrá resuelto no sólo el problema original, sino también otros muchos.

La generalización impulsa a compartir información disminuyendo, por consiguiente, la redundancia.

Un mecanismo de generalización propio de la orientación a objetos es la *herencia*, que permite compartir sin redundancias las características y comportamiento comunes a varias clases.

La herencia permite también definir nuevas clases a partir de otras ya existentes, de manera que presenten las características y comportamiento de éstas más, posiblemente, otras adicionales. Esta forma de emplear la herencia se conoce como *especialización*, ya que representa el proceso inverso a la generalización.

Polimorfismo

El polimorfismo[8] es la capacidad que puede poseer un componente para interpretar una solicitud de servicio de maneras distintas según el contexto

[8] Éste es uno de los conceptos más difíciles de definir, por la multitud de interpretaciones que se le han dado y de contextos en los que se emplea.

en el que se encuentra. Un componente tal se denomina *polimórfico*.

1.2.2. Objetos

No se dará una definición rigurosa de qué es un objeto, sino que nos aproximaremos informalmente a dicho concepto desde distintos puntos de vista:

Conceptual Un objeto es una entidad individual presente en el sistema que se está desarrollando, formada por la unión de un estado y de un comportamiento.

De la implementación Un objeto es una entidad que posee un conjunto de datos y un conjunto de operaciones que trabajan sobre ellos.

El estado de un objeto viene determinado por los valores que toman sus datos. Estos valores siempre han de cumplir las restricciones[9] que puedan haber sido impuestos sobre ellos.

Los datos se denominan también *atributos* y conforman la estructura del objeto. Las operaciones se denominan también *métodos* y representan los servicios que el objeto puede proporcionar.

Por ejemplo, una persona es un objeto del mundo real sobre la que se puede distinguir un conjunto de datos (nombre, apellidos, DNI, dirección, teléfono, etc.) que determinan su estado, junto con un conjunto de operaciones (cambiar de dirección o de número de teléfono) que determinan su comportamiento.

Podemos representar un objeto mediante un rectángulo que contiene su nombre subrayado como en la figura 1.2.

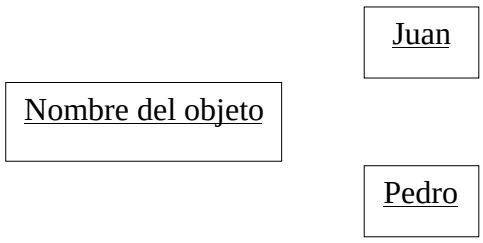

Figura 1.2: Representación de un objeto

Todo objeto presenta las tres características siguientes: un *estado*, un *comportamiento* y una *identidad*.

[9] Estas restricciones que deben cumplir todos los objetos de una misma clase se denominan *invariantes de clase*.

Estado Es el conjunto de valores de todos los atributos de un objeto en un instante de tiempo.

Cada atributo aporta información sobre el objeto que lo contiene, tomando un valor definido por un *dominio*. En la mayoría de los casos[10], un dominio es simplemente un conjunto de valores concretos representable por un tipo en el lenguaje de programación.

El estado de un objeto, por lo tanto, tiene un carácter *dinámico*, evoluciona con el tiempo, aunque ciertos componentes de él pueden permanecer constantes.

Comportamiento Agrupa todas las competencias de un objeto y queda definido por las operaciones que posee.

Las operaciones pueden limitarse a *observar* el estado interno del objeto o pueden *modificar* dicho estado. La evolución del estado de un objeto es consecuencia de la aplicación de sus operaciones. Éstas se desencadenan tras la recepción de un estímulo externo o *mensaje* enviado por otro objeto.

Las interacciones entre objetos se representan por medio de *diagramas de objetos* en los que los objetos que interactúan están unidos entre sí por trazos continuos llamados *enlaces*. Los mensajes «navegan» por los enlaces, en principio en ambas direcciones.

Identidad Caracteriza la propia existencia del objeto como ente individual. La identidad permite distinguir los objetos de forma no ambigua, independientemente de su estado.

De este modo es posible distinguir dos objetos en los que todos los valores de sus atributos sean iguales. Ambos objetos serían, en el sistema, lo que se denominan *clones*, uno réplica exacta del otro, pero poseerían su propia identidad.

La identidad es un concepto, no tiene por qué aparecer representada de manera específica en el modelo. Cada objeto posee su propia identidad de manera implícita. Desde el punto de vista de la implementación esto es aún más evidente: cada objeto ocupa sus propias posiciones de memoria.

La figura 1.3 representa en un *diagrama de objetos* varios clientes de un banco y distintos productos asociados con cada uno de estos clientes. Los segmentos que unen los objetos simbolizan los enlaces que existen entre un cliente y un producto particular.

[10] En general, un dominio puede venir dado por condiciones complejas o *restricciones* que permiten definirlo a partir de un superconjunto fácilmente caracterizable de éste.

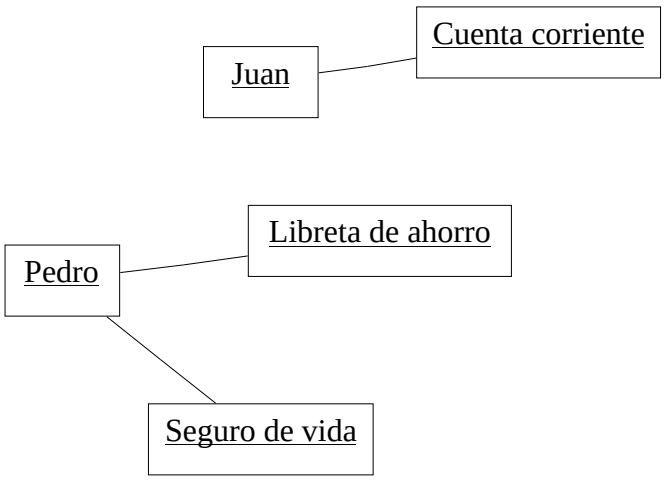

Figura 1.3: Representación de objetos relacionados entre sí

El comportamiento global de un sistema está determinado por la comunicación que se produce entre los objetos que lo componen. Es frecuente distinguir tres categorías de objetos según su comportamiento. Éstas se encuentran representadas en el diagrama de objetos de la figura 1.4.

Actores Son objetos que exclusivamente emiten mensajes. Por lo tanto, se comunican con otros objetos por iniciativa propia.

Servidores Son objetos que únicamente reciben mensajes. Su función es la de atender las solicitudes externas de otros objetos.

Agentes Son objetos que reúnen las características de los actores y de los servidores: pueden tanto emitir como recibir mensajes. En consecuencia, se comunican con otros objetos bien por iniciativa propia, bien debido a una solicitud externa.

Esta forma de modelar la interacción entre los objetos del sistema se denomina, por razones obvias, *paso de mensajes*. Cuando un mensaje se pasa desde un objeto emisor a un objeto receptor, éste es estimulado, produciéndose en él como reacción cierto comportamiento. Así, el paso de mensajes es lo que mantiene comunicados a los distintos componentes de un sistema orientado a objetos.

Este comportamiento puede provocar el envío de un mensaje de respuesta al objeto emisor, o a otros objetos, o no producir ningún mensaje sino sólo una modificación del estado interno del receptor o la observación de dicho estado

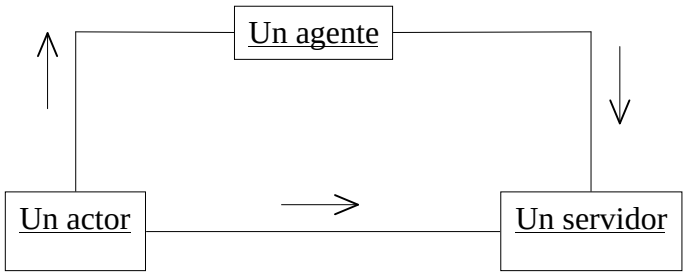

Figura 1.4: Categorías de objetos según su comportamiento

por parte del emisor. En definitiva, el comportamiento vendrá determinado por una operación.

De manera abstracta y general, un mensaje presenta la siguiente estructura:

$$nombre\ [destino,\ operación,\ parámetros]$$

donde *nombre* es el nombre del mensaje, *destino* es el objeto receptor, *operación* se refiere al método que se ejecutará tras la recepción del mensaje y *parámetros* representa la información que necesita dicho método para ejecutarse.

Los mensajes, como muestra la figura 1.5, se representan mediante flechas colocadas a lo largo de los enlaces que unen los objetos y permiten agrupar los flujos de control y de datos dentro de una única entidad.

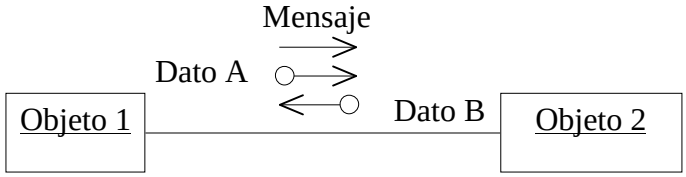

Figura 1.5: Representación de los flujos de control y de datos

1.2.3. Clases

Una clase es una descripción general que permite representar un conjunto de objetos similares. Por definición, todos los objetos que existen dentro de una clase comparten los mismos atributos y métodos. La clase encapsula las abstracciones de datos y operaciones que se necesitan para describir una entidad del mundo real.

Los atributos pueden ocultarse de manera que la única forma de operar sobre ellos sea a través de alguno de los métodos que proporciona la clase. Esto

permite ocultar los detalles de implementación, facilita el mantenimiento y la creación de componentes reutilizables.

Podemos representar una clase tal y como aparece en la figura 1.6, mediante un rectángulo que contiene su nombre y, opcionalmente, un compartimento para sus atributos y otro para sus métodos.

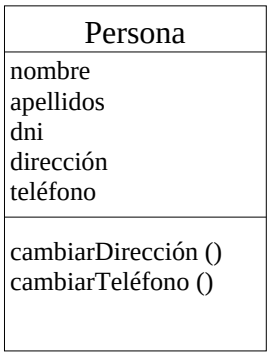

Figura 1.6: Representación de una clase

1.2.4. Relaciones entre clases

Los enlaces particulares que relacionan a los objetos entre sí pueden abstraerse en el mundo de las clases: a cada familia de enlaces entre objetos corresponde una *relación* entre las clases de estos mismos objetos.

Al igual que los objetos son *ejemplares de las clases*[11], los enlaces entre los objetos son *ejemplares de las relaciones* entre las clases.

Existen distintos tipos de relaciones entre clases. En concreto, distinguimos: la *dependencia*, la *asociación*, la *agregación* (en realidad, un tipo especial de asociación), la *generalización/especialización* y la *realización*.

La característica más interesante de las clases es que para su descripción no es preciso comenzar siempre partiendo de cero. Es posible basar la definición de una clase en las de otras utilizando las relaciones de agregación y de generalización/especialización.

En la figura 1.7 se muestran, empleando un *diagrama de clases* de ejemplo, las relaciones más comunes. Nótese que para distinguir los diferentes tipos de relaciones se emplean líneas que unen las clases relacionadas y símbolos adicionales: para representar la generalización/especialización se emplea un

[11] Se suele emplear también el término «instancia», traducción excesivamente literal del inglés.

triángulo, para la agregación, un rombo, y para la asociación, basta con la línea.

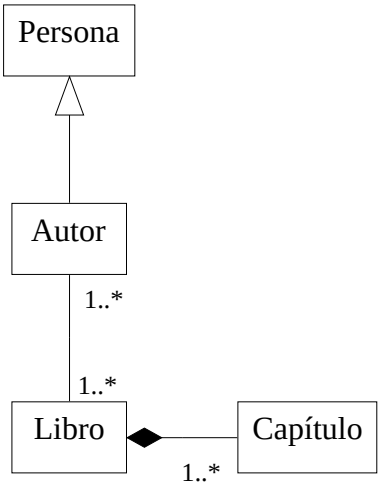

Figura 1.7: Relaciones entre clases

Dependencia

La dependencia es una relación de uso que declara que un cambio en la especificación de una clase puede afectar a otra que la utiliza, pero no necesariamente a la inversa.

Por ejemplo, una clase que recibe objetos de otra como parámetros de alguna de sus operaciones *depende* de ella.

Asociación

Una asociación expresa una conexión, en principio bidireccional, entre clases. Del mismo modo que las clases son abstracciones de los objetos, las asociaciones lo son de los enlaces que existen entre los objetos.

Las asociaciones poseen tres características principales:

Cardinalidad Indica el número de clases que intervienen en la asociación. Las cardinalidades más comunes son la *binaria* y la *ternaria*.

Multiplicidad Especifica el número de objetos que puede haber en cada extremo de la asociación. Así, una asociación puede ser *uno a varios*, *varios a uno*, *uno a uno*, etc.

Navegabilidad Determina el sentido en el que se puede recorrer la asociación. Así, las asociaciones pueden ser *unidireccionales* o *bidireccionales*.

Agregación

Permite expresar el hecho de que un objeto de una clase puede estar compuesto por objetos de otras. En realidad, no es más que una forma especializada de asociación. El tipo más común de agregación se denomina *composición* y permite especificar una relación *todo/parte*.

Generalización

La relación de generalización es la que se establece entre una clase y una o más versiones especializadas de ella. La que se está especializando se denomina *superclase* o *clase madre* y las versiones especializadas se denominan *subclases* o *clases hijas*.

Se dice que la subclase «hereda» de la superclase. Claramente, una clase puede pertenecer a la vez a ambas categorías.

Los datos y operaciones comunes a un grupo de subclases se colocan en la superclase para que así sean compartidos por todas las subclases.

De estas definiciones se deduce que es posible establecer una *jerarquía de clases* en la que los atributos y operaciones de la superclase son *heredados* por sus subclases que pueden añadir nuevos atributos y operaciones. Estas subclases pueden ser a la vez superclases de otras clases y así sucesivamente.

La herencia puede ser, principalmente, de dos tipos:

Simple Una subclase hereda de una única superclase.

Múltiple Una subclase hereda de más de una superclase.

Realización

La relación de realización permite modelar la conexión entre una *interfaz* y una clase que la implementa.

Al contrario que las clases convencionales, las interfaces no poseen datos; únicamente operaciones para las que, además, no proporcionan implementación. Una clase que «realiza» una o varias interfaces ha de proporcionar la implementación de todas sus operaciones.

Semánticamente, una relación de realización es una mezcla entre dependencia y generalización.

1.3. El entorno de desarrollo

Un *entorno de desarrollo de software* es un conjunto de herramientas que se emplean en el proceso de desarrollo de software. Esto incluye, en el sentido amplio del término, a la *plataforma de desarrollo*: es decir, el sistema operativo y el hardware empleado. Existen distintos tipos de entornos:

Tradicionales Constan de un conjunto de herramientas independientes que normalmente se usan en la etapa de implementación y en la de prueba. Ejemplos de estas herramientas son:

- Editores de código fuente
- Traductores (compiladores e intérpretes)
- Enlazadores
- Depuradores[12]
- Perfiladores
- Generadores de referencias cruzadas
- Sistemas de ayuda a la recompilación
- Sistemas de control de versiones
- Sistemas de generación de pruebas

Integrados Agrupan herramientas existentes en un entorno tradicional en una única herramienta. Suelen incluir sus propias bibliotecas de componentes reutilizables y también facilitar la creación de *interfaces gráficas de usuario*[13].

CASE El nombre proviene de sus siglas en inglés: *Computer Assisted Software Engineering* y han sufrido una gran evolución. En su estado actual, suelen ser entornos integrados que han sido extendidos para cubrir total o parcialmente el proceso de desarrollo y no exclusivamente la etapa de implementación. Generalmente facilitan el empleo de una metodología de desarrollo de software específica.

No nos dedicaremos aquí a explicar ningún entorno de desarrollo particular, ya que el objeto de nuestro estudio se centra exclusivamente en el lenguaje de programación C++.

[12] También llamados *trazadores* de código.
[13] IGU (en inglés, GUI: *Graphical User Interface*).

No obstante, se describirá en suficiente detalle cómo se produce un ejecutable a partir del código fuente, ya que este proceso es prácticamente común a todos los entornos y, aunque permanezca oculto al desarrollador, permite comprender mejor el porqué de ciertos errores de programación.

A estos efectos, supondremos que se emplea un entorno de desarrollo tradicional formado por un sistema operativo tipo UNIX, un editor y un compilador de C++. Concretamente, los autores han desarrollado los ejemplos sobre el sistema operativo LINUX, empleando el editor GNU Emacs y el compilador GNU C++ del proyecto GNU (*GNU's Not Unix*), de la FSF (*Free Software Foundation*).

1.3.1. Cómo dar la orden

El proceso de traducción nos lleva del fichero de texto con el *código fuente* al fichero binario con el *código ejecutable*. Cómo escribir correctamente el programa será objeto de estudio posterior. Por ahora baste suponer que ya se dispone del código fuente y que hay que obtener el ejecutable.

En todos los sistemas LINUX existe, al menos, un compilador de C++. Éste es, por lo general, el compilador GNU C++ y será el empleado en los ejemplos que aparecerán a continuación. Normalmente se ejecuta mediante la orden `c++`, o también `g++`.

El *fichero fuente* es un fichero de texto que crearemos con nuestro editor favorito. Supongamos que hemos creado nuestro primer programa en el fichero `hola.cpp`; la orden

```
% c++ hola.cpp
```

compilará este código para producir el ejecutable en el fichero `a.out`; para ejecutarlo simplemente escribiremos su nombre:

```
% a.out
¡Hola a todos!
```

Si ahora escribimos otro programa y damos la orden análoga para compilarlo, el nuevo fichero ejecutable `a.out` se escribirá sobre el antiguo, que se perderá. Para evitar esto el programa traductor, `c++`, dispone de una opción que nos permite darle otro nombre al fichero de salida. Esta opción es `-o` *fichero*; por lo tanto, otra forma de compilar es

```
% c++ -o hola hola.cpp
```

o bien

```
% c++ hola.cpp -o hola
```

En UNIX un fichero puede tener cualquier nombre y éste puede contener cualquier carácter salvo el separador de directorios, la barra `/` (y el `\0`). Es costumbre que el fichero ejecutable tenga el mismo nombre que el fuente, pero sin extensión, y el nombre del fichero con código fuente en C++ acabe en los caracteres `.cpp` (aunque también puede acabar en `.cc` o `.cxx`). Y recuerde que en UNIX las mayúsculas y las minúsculas se consideran caracteres diferentes.

1.3.2. Pasos en la traducción

En realidad, `c++` no es exactamente el compilador, aunque para abreviar digamos «el compilador `c++`». Más bien, es un programa *maestro* o *conductor* que simplemente se va a encargar de bifurcar procesos ejecutando una serie de programas, entre los que está el verdadero compilador, que son los que van a hacer realmente la tarea.

Además `c++` se encargará de gestionar las opciones de la línea de órdenes y sabrá a qué programa hay que pasárselas, creará y borrará ficheros intermedios temporales, etc.

Puede preguntarse por qué la traducción completa no se hace en un único programa. La respuesta es que es mucho más fácil para el fabricante hacerlo en pasos. Además el programa responsable de cada paso puede ser utilizado por separado y ser así aprovechado por otro compilador.

El esquema de la figura 1.8 indica de manera simplificada los pasos que se siguen en la traducción y las opciones que intervienen. Conceptualmente podemos imaginarnos que la traducción tiene lugar en esos pasos, aunque algunos fabricantes, por ejemplo, incluyan el preprocesador y el compilador en un mismo programa o se efectúe la compilación y la optimización en varias etapas.

Preprocesado El *preprocesador* es un programa que «prepara» el código fuente para el compilador. Nos ahorra muchísima tarea, pues gracias a él podemos incluir en el código otros ficheros (llamados *cabeceras*), definir símbolos que nos aclararán el programa (*macros*), compilar partes del código condicionalmente, etc. También se encarga de unir líneas partidas y cadenas de caracteres literales adyacentes, quitar los comentarios, sustituir *secuencias de escape* (véase 2.1), etc.

Las líneas que comienzan con el carácter *sostenido* (`#`) son interpretadas por el preprocesador, no por el compilador.

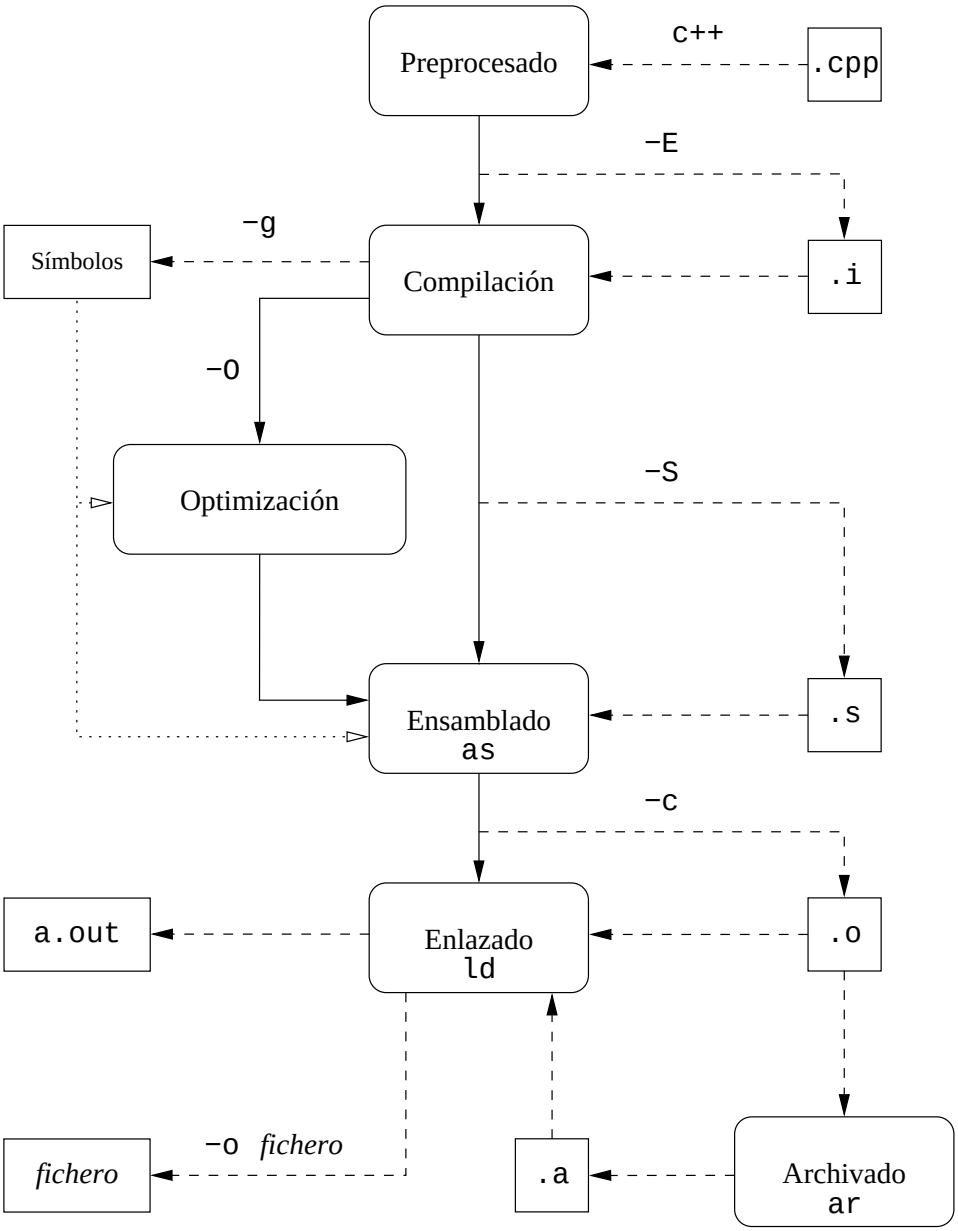

Figura 1.8: Pasos en la traducción de un programa

La opción `-E` preprocesa el código fuente y presenta el resultado en la salida estándar. Una extensión de fichero `.cpp` indica código fuente sin preprocesar, por lo que `c++` le pasa el preprocesador en primer lugar. Éste es el caso más normal.

Compilación El *compilador* propiamente dicho actúa ahora sobre el código preprocesado; en algunos sistemas el preprocesador y el compilador son un mismo programa, pero todo ocurre *como si* fueran dos distintos. La compilación puede suceder en un paso o en varios.

Dependiendo de las opciones pasadas a `c++` puede haber una[14] etapa de *optimización* de código, donde el compilador intenta eliminar posibles redundancias, almacenar ciertas variables en registros, etc., produciendo un mejor código ensamblador. Esto se consigue con la opción `-O` (*o* mayúscula), que puede ir seguida de un dígito para expresar distintos grados de optimización.

También es muy importante la opción `-g`, que hace que en el código resultante se incluya una tabla de símbolos que podrá ser interpretada más tarde por un programa llamado *depurador*, que nos permite controlar la ejecución del programa interactivamente y descubrir errores ocultos. No es recomendable mezclar las opciones `-O` y `-g`, y algunos compiladores no lo permiten siquiera.

Una extensión de fichero `.i` indica código fuente ya preprocesado, por lo que `c++` no le pasa el preprocesador, sino que lo compila directamente.

Ensamblado Aunque en algunos sistemas es normal que el compilador ya produzca código máquina directamente, otras veces hay otro paso intermedio y lo que se produce es código en ensamblador, que ya es un lenguaje de bajo nivel. Esto hace que el compilador sea más fácil de realizar, puesto que el *ensamblador* ya está hecho y se aprovecha. Además podemos ver el código ensamblador y modificarlo si sabemos y nos interesa afinar mucho.

Con la opción `-S` obtenemos un fichero de extensión `.s` con el código fuente en ensamblador. Asimismo, si a `c++` le pasamos un fichero con extensión `.s`, supone que contiene código ensamblador y le pasa el programa ensamblador directamente.

La opción `-c` deja el código máquina resultante del ensamblado en un fichero con extensión `.o` llamado *módulo objeto*, que ya es código máquina pero no es ejecutable aún. Esta opción es muy importante, pues nos permite la *compilación por separado*.

[14] O varias. La «optimización», más correcto sería decir «mejora», de código es un proceso complejo que puede requerir la realización de diversas tareas en distintos momentos de la traducción, incluso tras el ensamblado.

Enlazado En todos los sistemas existe una separación en este punto. El enlazado se efectúa aparte por otro programa. Éste se llama *enlazador* o *editor de enlaces*. Como su nombre indica, enlaza el módulo objeto producido por el ensamblador con otros módulos que le podamos indicar, y con los módulos correspondientes de las *bibliotecas* de funciones, que son archivos formados por módulos objeto con funciones ya compiladas. Sin que le digamos nada, `c++` se encarga de llamar al enlazador pasándole el archivo correspondiente a las funciones de la biblioteca estándar de C++ y otras muchas del sistema, de uso común.

El resultado de todo es el fichero `a.out` con el código máquina ejecutable. Ya se ha dicho que con la opción `-o` *fichero* podemos cambiarle el nombre.

1.3.3. Un primer programa

Hemos compilado el programa que guardábamos en el fichero `hola.cpp`. Aunque este programa es tremendamente sencillo, ahora vamos a ver qué tiene dentro, diseccionándolo para mostrar algunos conceptos elementales. A continuación se presenta el listado:

hola.cpp

```
1  // El programa que saluda
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8    cout << "¡Hola a todos!" << endl;
9  }
```

Escritura libre En C++, al igual que en C y que en la mayor parte de los lenguajes modernos, la escritura es «de formato libre». Esto quiere decir que podemos utilizar los blancos (espacios, tabuladores, saltos de línea o saltos de página) a discreción. Con que no cortemos por la mitad un *lexema*, podemos separarlos por la cantidad de blancos que queramos; esto significa que podemos escribir los programas de forma agradable o ilegible. Unas mínimas normas de estilo ayudan a mantener la legibilidad del código fuente.

¿Y qué es un lexema? Por ahora baste decir que un lexema es un operador como `<<`, una palabra reservada del lenguaje como `if` o `while`, un identificador o nombre de función o variable como *main* o *cout*, un

signo delimitador como `{` o `;`, una constante como `0` ó `0.25e-10` ó una cadena de caracteres literal como `"¡Hola a todos!"`.

Comentarios Todos los caracteres que aparecen después de `//` constituyen un *comentario*, hasta el fin de la línea.

Directrices del preprocesador Las líneas que comienzan con el carácter `#` (quizá rodeado de blancos, aunque por historia y estilo se pone siempre ese carácter el primero de la línea) son interpretadas por el preprocesador, no por el compilador. La palabra reservada del preprocesador que viene a continuación se llama *directriz* o directiva del preprocesador.

En este caso utilizamos la directriz `include` seguida del nombre de un fichero entre ángulos. Éstos le indican al preprocesador que busque ese fichero en ciertos directorios; el fichero en cuestión se llama «de cabecera» (suelen llevar la extensión `.h`, por el inglés *header*, salvo las cabeceras estándar) y contiene ciertas declaraciones útiles. En este caso, el fichero se llama `iostream`, y contiene declaraciones que se necesitarán más adelante.

Espacios de nombres Un espacio de nombres es un mecanismo para agrupar lógicamente un conjunto de identificadores (nombres de tipos, de funciones, etc.).

Existe un espacio de nombres global y bajo él se define el resto de los espacios de nombres. A un identificador que se encuentra en un espacio de nombres se puede acceder de distintas formas.

En este caso la cláusula `using` está cargando el espacio de nombres estándar (*std*) en el espacio de nombres actual, que es el global, permitiendo así el empleo de los identificadores *cout* y *endl* (declarados en `<iostream>`) sin necesidad de cualificación.

Funciones Una función es un fragmento de código que realiza cierta tarea. Se comunica con el resto del programa mediante los parámetros que recibe, y mediante lo que devuelve. Como en C++ no existe el «programa principal», debe haber una función que sea la primera que se ejecute y llame a todas las demás. Es el enlazador quien se encarga de buscarla, y busca una que se llame *main*, que en inglés significa «primera» o «principal»[15].

Así que en todo programa C++ debe haber al menos una definición de función, y una y sólo una de las funciones que contenga el programa debe llamarse con el nombre *main*.

[15] A diferencia de C, es posible, bajo determinadas circunstancias, que ésta no sea realmente la primera función en ejecutarse.

Para definir una función pondremos primero el *tipo de datos* del valor que devuelve. En este caso la palabra reservada `int` indica que es un dato de tipo numérico entero (*integer*). En C++ los procedimientos son simplemente funciones que no devuelven nada. En tal caso se pondría como tipo de retorno la palabra reservada `void`, que en inglés significa «vacío».

A continuación viene el nombre de la función y, entre paréntesis, los parámetros formales separados por comas. Los paréntesis son siempre obligatorios; si una función no recibe nada, como en nuestro caso, en la definición pondremos los dos paréntesis solamente o bien la palabra reservada `void`.

Por último viene el *cuerpo* de la función, que es una *instrucción compuesta*; esto es, una serie de *instrucciones* encerradas entre llaves. Una instrucción especifica qué cómputo o acción se debe efectuar.

Una función acaba normalmente cuando se llega a la llave de cierre de su cuerpo o cuando se encuentra `return`. Esta palabra reservada, que en inglés significa «retornar», indica que la función ha acabado y el control debe transferirse al punto de llamada, posiblemente devolviendo un valor. Si la función no devuelve nada, tras la palabra `return` se pondrá el punto y coma de fin de instrucción. Pero si debe devolver un valor, se pondrá una *expresión* del tipo correspondiente.

En el caso especial de la función *main()*, el compilador inserta automáticamente `return 0;` antes de su llave de cierre.

Bien, pero si *main()* es la función principal no llamada por ninguna otra, ¿a quién le devuelve ese cero? La respuesta es «al entorno que ha llamado al programa»; es decir, en el caso de los sistemas operativos tipo Unix, al *shell* o intérprete de órdenes[16]. El cero, por convenio, le indica que el programa ha acabado bien; otro número indicaría que ha ocurrido algún tipo de error.

Por ejemplo, supongamos que un programa debe leer un fichero y éste no existe. Entonces podría terminarse con una instrucción `return` pero devolviendo el valor 1, y desde el *shell* podríamos comprobar ese código de retorno y saber qué ha pasado.

Salida de datos Para imprimir se utiliza un operador, `<<`, que se lee como *poner en* o *inserción*.

Aquí ya vemos algo interesante: en C (y también en C++, ojo) el operador `<<` es el de desplazamiento de bits a la izquierda, pero en `<iostream>` este operador ha sido *sobrecargado*, vale decir redefinido, para darle otro significado. ¿Acaso ha perdido el anterior? No, pero el

[16] Observe bien que esto no implica que estemos imprimiendo nada en ningún sitio.

empleo de uno u otro se determina por el contexto; en este caso, por el tipo de los operandos.

El identificador *cout* representa a un objeto asociado al flujo de datos de salida; podríamos decir que es el equivalente al *stdout* de C; este objeto y la sobrecarga de `<<` necesaria están definidos en la cabecera `<iostream>`.

Además, se puede aplicar consecutivamente el operador sobre un mismo objeto. Por ejemplo, se podría haber hecho:

```
cout << "¡Hola" <<
     << " " << "a" << " "
     << "todos!" << endl;
```

O sea, cada vez que se emplea el operador `<<`, la impresión continúa por donde se quedó. El identificador *endl*, que es un *manipulador*, provoca un salto de línea seguido de un «vaciado» (*flush*) de los datos del búfer de salida.

Cadenas literales Como en C, las cadenas de caracteres literales son secuencias de caracteres entre comillas dobles. Éstas pueden partirse por donde queramos, que el preprocesador las concatenará.

Si una cadena literal es demasiado grande para caber en una línea del listado, tenemos varias posibilidades; a continuación las enumeramos de peor a mejor; así que procure siempre que pueda utilizar la última. En todos los casos se va a imprimir exactamente lo mismo:

```
En C++ las cadenas literales se pueden partir.
```

1. Escribir la línea de todas formas, si el editor nos lo permite. Esto es lo peor, porque a la hora de sacar un listado puede que no veamos parte de la línea.
2. Si la cadena cabe sola en una línea, podemos escribirla en ella; así se pierde el sangrado y el efecto estético no es muy bueno:

   ```
   cout <<
    "En C++ las cadenas literales se pueden partir."
     << endl;
   ```

3. Es posible cortarla por cualquier sitio con una barra invertida seguida inmediatamente de un retorno de carro. Tampoco esta vez queda bien:

   ```
   cout << "En C++ las cadenas \
   literales se pueden partir." << endl;
   ```

4. Por último, podemos escribir varias cadenas literales separadas por blancos, y el preprocesador las concatenará:

   ```
   cout << "En C++ las cadenas "
           "literales se pueden partir." << endl;
   ```

1.4. Diferencias con C

Uno de los objetivos que tuvo en mente Bjarne Stroustrup cuando diseñó el lenguaje C++ fue la compatibilidad con C, para que los programadores que ya conocieran este lenguaje pudieran aprenderlo con más facilidad. De hecho, el comité ANSI que estandarizó el lenguaje C en 1989 tomó prestadas muchas cosas de C++: los prototipos de funciones, las constantes, etc. El hecho es que C++ actualmente es casi compatible con C ANSI/ISO, de forma que prácticamente todos los ejemplos del conocido libro de Brian W. Kernighan & Dennis M. Ritchie [6] pueden compilarse con un compilador de C++.

Sin embargo, C++ no acepta aspectos del C tradicional. Por ejemplo, los prototipos son obligatorios, es decir, todas las funciones han de ser declaradas antes de ser llamadas.

C++ es un compromiso entre el bajo y el alto nivel. Compatible con la forma moderna (ISO) de C en un gran porcentaje, retiene su capacidad que le acerca a lenguajes ensambladores: operadores y campos de bits, uniones, etc. El programador puede escribir código al nivel apropiado al problema mientras sigue manteniendo contacto con los detalles a un nivel más cercano a la máquina.

A continuación vamos a ver aspectos comunes de C y C++ que son tratados en C++ de forma diferente. En general, podemos decir que C++ lleva a cabo más comprobaciones que C, y es más estricto. Normalmente el programa `lint`[17] no haría falta en C++, pues el compilador se encarga de su tarea.

1.4.1. Palabras reservadas

C++ proporciona muchas más palabras reservadas que C. En la tabla 1.1 pueden verse todas ellas, apareciendo subrayadas las que son también de C.

Algunas de las palabras reservadas de C++ son en C macros o tipos definidos en las cabeceras de la biblioteca estándar. Puede verlas en la tabla 1.2.

1.4.2. Cabeceras estándar

A diferencia de lo que ocurre en C, las cabeceras estándar de C++ no llevan el sufijo `.h`, ni ningún otro. De hecho, ni siquiera existe la obligación de que el sistema las guarde como ficheros.

[17] El programa `lint` analiza un programa en C haciendo comprobaciones exhaustivas que el compilador de C no hace.

and	and_eq	asm	auto
bitand	bitor	bool	break
case	catch	char	class
compl	const	const_cast	continue
default	delete	do	double
dynamic_cast	else	enum	explicit
export	extern	false	float
for	friend	goto	if
inline	int	long	mutable
namespace	new	not	not_eq
operator	or	or_eq	private
protected	public	reinterpret_cast	register
return	short	signed	sizeof
static	static_cast	struct	switch
template	this	throw	true
try	typedef	typeid	typename
union	using	unsigned	virtual
void	volatile	wchar_t	while
xor	xor_eq		

Cuadro 1.1: Palabras reservadas de C++

Además, las cabeceras que ya existían en C cumplen también esta regla, si bien sus nombres se preceden del prefijo `c`. Algunas son de muy poco uso en C++, por disponerse de herramientas más potentes a las proporcionadas por ellas. Así, la `<cstdio>` de C++ equivale a la `<stdio.h>` de C; esta cabecera se emplea muy poco, ya que `<iostream>` la supera ampliamente.

1.4.3. Tamaño de las constantes literales de carácter

En C una constante literal de carácter, como `'A'`, se trata como un `int`; es decir, se cumple:

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	wchar_t	xor	xor_eq

Cuadro 1.2: Palabras reservadas que son macros o tipos definidos en cabeceras de C

```
sizeof 'A' == sizeof(int)
```

Sin embargo, en C++, tal constante es de tipo `char`; o sea,

```
sizeof 'A' == sizeof(char) == 1
```

1.4.4. Entrada y salida de datos

Ya se ha visto el famoso programa que saluda en C++. Si se observa ahora el programa tradicional escrito en C, se comprueba que éste ¡también es un programa válido en C++!

hola.c

```
1 /* El programa que saluda */
2
3 #include <stdio.h>
4
5 int main()
6 {
7   printf("¡Hola a todos!\n");
8   return 0;
9 }
```

Al comparar ambos, lo primero que se observa es el cambio en la forma de poner el comentario. En efecto, en C++, si bien podemos usar aún la forma tradicional de C, se emplea un nuevo estilo de comentario.

También se observa que ahora no se incluye el fichero de cabecera `<stdio.h>` sino `<iostream>`, que pone a nuestra disposición la posibilidad de realizar de E/S a través de un modelo de flujos (*streams*) orientado a objetos.

Se ve a continuación que, en efecto, la forma de escribir en la salida estándar no es la misma que antes; ya no usamos una función de la biblioteca estándar, como *printf()*, sino un operador, `<<`. Éste se aplica a *cout* para producir la salida de datos y recibe el nombre de *poner en* o *inserción*.

El objeto *cout* es el flujo estándar de salida de datos. Éste tiene su contrapartida en *cin*: el objeto que representa el flujo estándar de entrada de datos.

Para leer datos de la entrada estándar se emplea el operador `>>`, que ha sido sobrecargado análogamente a `<<` de forma que utilizado así sobre *cin* se llama *coger de* o *extracción*.

Para comprobar que el valor leído ha sido válido (es decir, que tiene el formato apropiado al tipo requerido y no ha habido errores de lectura), basta emplear el objeto en cualquier expresión lógica después de realizar la lectura.

Observamos que estos operadores efectúan automáticamente las conversiones necesarias de los valores que tratan (están sobrecargados para manejar distintos tipos de datos), de manera que no tenemos que preocuparnos por los formatos, al contrario que con *printf()* y *scanf()*.

Ejemplo:

El siguiente programa va solicitando distancias en millas y las transforma a kilómetros mientras los datos introducidos sean correctos.

mi-km.cpp

```
1  // Convierte de millas a kilómetros
2
3  #include <iostream>
4  using namespace std;
5
6  const double m2k = 1.609;
7  inline double conversion(double mi) { return mi * m2k; }
8
9  int main()
10 {
11   do {
12     cout << "Distancia en millas: ";
13     double millas;
14     if (cin >> millas)
15       cout << "La distancia es " << conversion(millas) << " km."
16            << endl;
17   } while (cin);
18 }
```

Obsérvese cómo *cin* se emplea en la condición del bucle. Allí se transforma automáticamente en un valor booleano que indica si la última lectura realizada tuvo éxito.

También, en lugar de usar una macro como

```
#define M2K 1.609
```

se usa una constante.

Del mismo modo no se emplea algo como

```
#define CONVERSION(m) ((m) * 1.609)
```

porque la palabra reservada `inline` evita el principal inconveniente de las funciones sobre las macros: la eficiencia.

Por último, hay que decir que las variables se declaran y definen como en C, salvo que además pueden definirse en cualquier sitio, no sólo al principio de un bloque.

1.4.5. Comentarios

C++ entiende, por compatibilidad, el estilo de comentario de C, de todos conocido, pero introduce un nuevo símbolo de comentario, la doble barra `//`. Cualquier cosa tras esta doble barra en una misma línea es tratada como comentario, salvo que el símbolo forme parte de una cadena de caracteres (o de un *carácter multi-byte*, que es más raro) o que ya esté dentro de un comentario.

Normalmente se prefiere esta forma a la anterior, pues es más fácil de escribir y uno no se olvida del cierre del comentario. Sin embargo, esto puede, en ocasiones, causar problemas.

Ejemplo:

```
#define MAX 9              // tamaño máximo
...
for (i = 0; i < MAX; i++)
  ...
```

Si el preprocesador sustituye los comentarios por un espacio en blanco, como debería, no hay problema; pero si el preprocesador deja los comentarios para que sea el compilador quien los quite, lo cual, aunque raro, puede suceder en algún sistema, el compilador recibe:

```
for (i = 0; i < 9              // tamaño máximo; i++)
  ...
```

que, evidentemente, produce un error, que no ocurriría con el viejo estilo de comentario.

Aparte de estas rarezas, lo normal es usar siempre este tipo de comentario. Una línea como:

```
cout << "El comentario es // en C++" << endl;   // esto funciona
```

no causa ningún tipo de problema.

El nuevo estilo de comentarios está bien para estos casos, pero el tradicional se suele emplear cuando éstos ocupan varias líneas completas y también para anular fragmentos de código (siempre que no se aniden).

1.4.6. Empleo de macros

Como se sabe, el empleo de las macros del preprocesador tiene algunas desventajas; C++ intenta solventarlas.

El problema radica en que el preprocesador es un programa aparte que no sabe nada acerca de la sintaxis del lenguaje C: funciona como una herramienta de sustitución textual.

Funciones «en línea»

Supongamos que hemos definido la siguiente macro para hallar el cuadrado de un número:

```
#define SQR(x) x * x
```

Si la empleamos de esta manera:

```
i = SQR(a + b);
```

el código se expande a

```
i = a + b * a + b;
```

que es igual a $ab+a+b$, no a $(a+b)(a+b)$. Esto se soluciona con paréntesis:

```
#define SQR(x) ((x) * (x))
```

pero esto aún no nos defiende contra la llamada `SQR(a++)`, que se expande textualmente a `((a++) * (a++))`, con lo que a se incrementa dos veces.

El uso de una función arregla esto, pero tiene un inconveniente: es mucho más lento llamar a una función que emplear una macro, que se expande en tiempo de preprocesado. Esto es lo que remedia C++ mediante la palabra reservada `inline`:

```
inline int sqr(int x) { return x * x; }
```

Con esta palabra se hace una petición al compilador para que expanda el código de la función en cada llamada, evitando así el trabajo de una llamada verdadera a función. No obstante se conservan las características de una función: paso de parámetros, comprobación de tipos, etc.

El compilador puede incluso no hacer caso de esto si ve que no es conveniente, quizá porque la función sea muy grande. Esta palabra sólo debe emplearse para funciones muy cortas. También es frecuente que un buen compilador al optimizar detecte automáticamente qué funciones deben ser tratadas como `inline`, independientemente de que se haya puesto esta palabra o no. Ocurre como con el modificador de tipo `register` (véase §2.2.2).

> Esta característica ya viene incorporada en el compilador GNU C y en otros muchos; quizá en la próxima revisión del estándar ISO se adopte en C.

Constantes

Las constantes fueron recogidas con algunas diferencias por el estándar ANSI/ISO de C, si bien quizá aún no se utilizan mucho, quizá porque el C tradicional no lo admitía o por falta de costumbre.

Las constantes en C tienen enlace externo, mientras que en C++ el enlace es interno (véase §2.2.2). Como consecuencia, en C++ no hay ningún problema en definir constantes en ficheros de cabecera.

En lugar de usar el preprocesador para definir constantes simbólicas, se emplea el modificador de tipo `const` para definir un objeto[18] que no puede cambiar una vez inicializado; dicho de otra forma, no puede usarse en la parte izquierda de una asignación.

Un *valor-i* (*lvalue* en inglés, o sea, «valor izquierdo»), es una expresión que puede utilizarse como una dirección donde se puede almacenar algo. Una variable usada en la parte izquierda de una asignación es un valor-i donde se está guardando algo, como en `i = 2;`. Pues bien, un objeto constante es un *valor-i no modificable*. Esto implica que debe ser inicializado.

[18] No nos referimos aquí a un objeto en el sentido que se le da en POO, sino a un objeto de datos; llamar *variable* a una constante sería un contrasentido.

Ejemplo:

```
const size_t n;              // ERROR en C++, bien en C (n es 0)
const size_t n = 100;        // bien
double v[n];                 // bien en C++, ERROR en C

n = 200;                     // ERROR: n es constante
++v;                         // ERROR: v no es modificable

const char* s = "hola";      // puntero a «const char»

s = "adiós";                 // bien: s es modificable
s[0] = 'A';                  // ERROR: *s es constante

const char* const t = s;     // puntero constante a «const char»

t = "adiós";                 // ERROR: t no es modificable
t[0] = 'A';                  // ERROR: *t es constante
```

1.4.7. Declaraciones

Enumeraciones

El tipo de datos enumerado fue añadido en C a principios de la década de los 80, incluso Brian W. Kernighan & Dennis M. Ritchie escribieron un añadido a la primera edición de su libro para incluirlo. Pero C trata una variable enumerada como de tipo `int`. Ni siquiera se comprueba que sólo tome los valores de la enumeración, aunque un buen compilador puede avisar de ello. En cambio, en C++ el tipo enumerado es un tipo distinto del entero.

Las constantes de la enumeración, también se denominan *enumeradores*, se consideran como constantes enteras (`const int`), y son otra alternativa a las constantes simbólicas del preprocesador, o macros. Por omisión, los valores de los enumeradores se asignan en orden creciente a partir de 0.

Ejemplo:

En C++, supuesta la declaración

```
enum estado_t { soltero, casado, separado, divorciado, viudo };
```

no tiene por qué ser cierto `sizeof(enum estado_t) == sizeof(int)`.

En todo caso, se cumple que `soltero == 0`, `casado == 1`, `separado == 2`, `divorciado == 3` y `viudo == 4`.

Cada enumeración es un tipo distinto. El tipo de un enumerador es su enumeración. Siguiendo con el ejemplo, *casado* es de tipo *estado_t*.

Declarar una variable enumerada en lugar de una `int` da al compilador información sobre su utilización. Con la enumeración del ejemplo, el compilador podría emitir una advertencia en el caso de que sólo se manejen cuatro de los cinco valores de *estado_t* en una instrucción `switch` (véase §3.3.3).

Un enumerador puede ser inicializado por una expresión constante de tipo booleano, carácter o entero.

Un valor de tipo booleano, carácter o entero puede ser convertido explícitamente a un tipo de enumeración. El resultado de tal conversión no está definido a menos que el valor esté dentro del rango de la enumeración.

No hay conversión implícita de un entero a una enumeración, ya que la mayor parte de los valores enteros no están representados en una enumeración concreta.

El `sizeof` de una enumeración es el `sizeof` de algún tipo booleano, carácter o entero que pueda contener su rango y no sea mayor que `sizeof(int)`, a menos que un enumerador no pueda ser representado como `int` o `unsigned int`.

Por omisión, las enumeraciones se convierten a enteros en las operaciones aritméticas.

Rótulos como nombres de tipos

En C los rótulos, o nombres de una enumeración, estructura o unión, no son nombres de tipos.

Ejemplo:

```
enum palos { oros, bastos, copas, espadas };
struct cartas { enum palos p; int valor; };

struct cartas baraja[40];
```

Observe que tenemos que emplear las palabras `enum` y `struct`. Una alternativa mejor.

Ejemplo:

```
typedef enum palos { oros, bastos, copas, espadas } palos;
typedef struct cartas { palos p; int valor; } cartas;

cartas baraja[40];
```

Pero en C++ los nombres o rótulos de enumeraciones, estructuras, clases y uniones son nombres de tipos, por lo que `typedef` no es necesario para esto:

Ejemplo:

```
enum palos { oros, bastos, copas, espadas };
struct cartas { palos p; int valor; };

cartas baraja[40];
```

Lugar de las declaraciones

En C las declaraciones de objetos tienen que estar al principio de un bloque; es decir, inmediatamente detrás de la llave de apertura. Sin embargo en C++ pueden ir cerca de donde se utilicen, haciendo el código algo más legible. El alcance es el bloque más interno donde estén, y la visibilidad, a partir de donde se declaren.

En realidad el compilador reserva el espacio para los objetos a la entrada del bloque, como en C, sólo que impide que su nombre sea utilizable hasta que se produce la declaración.

Uno de los casos más comunes es el de las variables con ámbito de `for` (véase §3.4.1).

Ejemplo:

La siguiente función baraja una ídem. Se utilizan las declaraciones del ejemplo anterior.

```
void barajar(cartas mazo[])
{
  for (int i = 0; i < 40; ++i) {
    int k = rand() % 40;  // escoge carta al azar
```

```
    carta t = mazo[i];    // intercambia 2 cartas
    mazo[i] = mazo[k];
    mazo[k] = t;
  }
}
```

Aquí la variable i sólo existe dentro del bucle: tiene ámbito de `for`.

Empleo de `void`

La palabra reservada `void` fue introducida en algunos compiladores de C a principios de los 80, y fue rápidamente adoptada por el comité ANSI. Tiene varios significados, aunque las diferencias existentes entre C y C++ se reducen a los siguientes casos:

1. Cuando se emplea para declarar una función que no va a recibir parámetros.

   Para fijar ideas supongamos que el nombre de la función es *f()* y que devuelve un entero.

   En C puede declararse de la forma antigua: `int f();` o de la moderna: `int f(void);`. No obstante, la primera indica al compilador que no se sabe nada de los parámetros; la segunda le informa de que la función no recibe ninguno.

   En C++ es obligatorio declarar la función antes de llamarla, y con el prototipo, por lo que `int f();` significa otra cosa: que no recibe parámetros; o sea, lo mismo da `int f();` que `int f(void);`.

   En la definición de la función, es decir, donde se escribe su cuerpo, este empleo de `void` es opcional (tanto en C como en C++), pues no hay ambigüedad.

   En conclusión, la palabra reservada `void` se suele emplear en C para denotar la ausencia de parámetros, siendo innecesaria para este propósito en C++.

2. Cuando se emplea para declarar un *puntero genérico*: `void*`.

   En C, un puntero cualquiera puede ser convertido implícitamente en puntero genérico y viceversa.

   En C++, que tiene un sistema de tipos más rígido, se puede convertir implícitamente cualquier puntero en un puntero genérico, pero no al revés. Esto exige una conversión explícita.

Tanto en C como en C++ los punteros genéricos no pueden ser desreferenciados, pues no tiene sentido, ya que contienen direcciones puras y no se dispone del tamaño del posible objeto al que apuntan.

Ejemplo:

```
int e;

void* pg;       // puntero genérico
int* pe;        // puntero a entero

pg = &e;        // bien
pe = pg;        // ERROR en C++, bien en C
pe = (int*)pg;  // bien

e = *pe;        // bien
e = *(int*)pg;  // bien
e = *pg;        // ERROR: no se puede desreferenciar
```

Prototipos

En C++ no se admite la forma antigua o tradicional de C de declarar las funciones, ni de definirlas. Además deben ser declaradas explícitamente, el compilador no lo hará por nosotros.

Ejemplo:

```
double f();                 /* C: parámetros desconocidos */
double f();                 // C++: sin parámetros
double f(void);             // C y C++: sin parámetros
double f(...);              // C y C++: parámetros desconocidos
double sqrt(double x);      // bien en C y en C++
double sqrt(double);        // bien en C y en C++ (sin nombre)
int scanf(const char*, ...); // Nº variable de parámetros
int scanf(const char* ...);  // igual: ERROR en C y bien en C++
```

Con el prototipo anterior de *sqrt()* la llamada

```
d = sqrt(4);
```

hace que el valor *int* `4` se convierta en el *double* `4.0` antes de la llamada.

Los puntos suspensivos (elipsis) en una función indican explícitamente que no sabemos nada de sus parámetros, como es el caso de la función de la biblioteca estándar de C *scanf()*, de la cual sólo conocemos con seguridad su primer parámetro. Como vemos en el ejemplo, en C++ no hace falta la coma tras el último parámetro conocido. Para escribir una función con número variable de parámetros tenemos que usar las macros definidas en la cabecera estándar `<cstdarg>`. En C++ esto se usa muy poco.

1.4.8. Definiciones de funciones

Las definiciones de función al estilo tradicional de C no son admitidas por un compilador de C++.

Ejemplo:

El siguiente esqueleto no es correcto en C++, aunque es perfectamente válido en C ISO (por compatibilidad con el C tradicional).

```
int copia(s, t, n)
char *s, *t;            /* int n; */
{
  ...
}
```

En su lugar debe emplearse la forma moderna, no admitida por el C tradicional:

```
int copia(char* s, char* t, int n)
{
  ...
}
```

C++ permite que una definición de función no especifique el nombre de un parámetro. Estos parámetros anónimos son útiles en ocasiones.

Ejemplo:

```
...
#include <csignal>

void gestor_SIGINT(int)          // parámetro anónimo
{
```

```
  exit(0);
}

int main()
{
  signal(SIGINT, gestor_SIGINT); // requiere un void (*)(int)
  ...
}
```

1.4.9. Uniones anónimas

Esto no existe en C, ni tradicional ni ANSI/ISO. No obstante, es un detalle menor de poca utilidad.

Una unión anónima no tiene nombre, o rótulo; la novedad es que ahora tampoco hace falta definir variables de esa unión. C++ permite utilizar sus miembros directamente, siempre que no haya ambigüedad; es decir, que no haya otras variables con esos nombres.

Ejemplo:

El siguiente programa muestra la representación binaria de un número de coma flotante y doble precisión que se lee de la entrada estándar.

union.cpp

```
1  #include <iostream>
2  #include <climits>
3  using namespace std;
4
5  inline void mostrar_bits(unsigned char byte)
6  {
7    for (size_t i = CHAR_BIT; i--;)
8      cout << ((byte >> i) & 1);
9  }
10
11 int main()
12 {
13   const size_t n = sizeof(double);
14   union {
15     unsigned char byte[n];
16     double d;
17   };
18
```

```
19    cout << "Introduzca un número: ";
20    cin >> d;
21    for (size_t i = n; i--;)
22      mostrar_bits(byte[i]);
23    cout << endl;
24  }
```

Si la unión anónima es global, obligatoriamente debe tener clase de almacenamiento estática (véase §2.2.2).

1.4.10. Estilo

El estilo de escritura en C++ es el mismo prácticamente que en C, respecto al sangrado, espaciado, etc. Una excepción es que a la hora de declarar punteros el operador `*` se pone junto al tipo base, no junto al nombre. Lo mismo con las referencias (véase §1.5.2), aunque en C no existen:

```
int *a;    /* estilo de C */
int* a;    // estilo de C++
int& b;    // estilo de C++, en C no existen referencias
```

Si emplea un buen editor, como Emacs, la escritura del programa le será más cómoda. Por ejemplo, al editar un fichero cuyo nombre acaba en `.cpp`, Emacs se coloca automáticamente en «modo C++»: los elementos sintácticos pueden aparecer en distintos colores, el tabulador sangra la línea adecuadamente, las llaves y los dos puntos son «eléctricos» (se ponen automáticamente en su sitio), etc.

1.5. Recorrido por el lenguaje

1.5.1. Espacios de nombres

Un *espacio de nombres* es un mecanismo para agrupar lógicamente un conjunto de identificadores (nombres de tipos, de funciones, etc.). Los espacios de nombres permiten mantener separados lógicamente grandes componentes de software (como las bibliotecas) de manera que no interfieran.

Existe un espacio de nombres global, que no posee nombre, y bajo él se definen todos los objetos, incluidos los restantes espacios de nombres.

Un espacio de nombres se crea con un bloque `namespace`. Por ejemplo, el siguiente fragmento sirve para declarar una serie de primitivas gráficas bajo un espacio de nombres llamado *Graficos*:

```
namespace Graficos {
  void punto(int x, int y, int c);
  void circulo(int x, int y, int r, int c);
  void rectangulo(int x0, int y0, int x1, int y1, int c);
  ...
};
```

Existe un espacio de nombres estándar, llamado *std*, en el que se encuentra la biblioteca estándar.

Cuando un identificador se encuentra en un espacio de nombres distinto al de trabajo, es necesario un mecanismo que nos permita acceder a su nombre. Esto puede hacerse de distintas maneras:

1. Cargando el espacio de nombres completo al que pertenece en el espacio de nombres en el que estamos trabajando.
2. Cargando sólo el nombre concreto al que se desea acceder.
3. Cualificando el identificador con el nombre del espacio al que pertenece.

Ejemplo:

Para cargar completamente el espacio de nombres estándar se hace:

```
using namespace std;
```

y si lo único que se desea utilizar es *cout* y *endl*, puede escribirse:

```
using std::cout;
using std::endl;
```

Nótese que esto no declara los objetos *cout* y *endl*: previamente se tiene que haber incluido `<iostream>`.

Por último, es posible cualificar el identificador:

```
std::cout << "¡Hola a todos!" << std::endl;
```

Al operador `::` se le denomina *operador de resolución de ámbito*.

Los espacios de nombres se pueden anidar, aunque esto sólo suele ocurrir con componentes de software de gran entidad.

1.5.2. Referencias

C++ posee un tipo de datos que, en cierto modo, complementa la funcionalidad de los punteros de C: el tipo de las referencias. Una referencia es una especie de «puntero encubierto» y su principal aplicación radica en permitir el paso y devolución de parámetros por referencia.

Ejemplo:

El ejemplo clásico es la función que intercambia los valores de dos variables de un cierto tipo.

```
void intercambiar(int& a, int& b)
{
  int t = a;
  a = b;
  b = t;
}
```

Nótese que la sintaxis de declaración es análoga a la de los punteros, pero sustituyendo `*` por `&`.

Hay ocasiones en las que conviene emplear referencias, no para modificar los parámetros reales, sino para ganar eficiencia. En tal caso, y puesto que no se desea realmente una modificación, se declaran dichas referencias constantes.

1.5.3. Gestión de la memoria dinámica

La memoria dinámica se gestiona en C++ a través de los operadores `new` y `delete`. Éstos sustituyen con notables ventajas a las funciones *malloc()*, *calloc()* y *free()* heredadas de C.

Ejemplo:

El siguiente fragmento crea un vector dinámico:

```
size_t n;

cout << "n = "; cin >> n;
double* v = new double[n];
```

y este otro lo destruye:

```
delete []v;
```

Al disponer de una biblioteca estándar tan potente, se evita en la mayoría de las aplicaciones el tener que gestionar la memoria dinámica directamente.

1.5.4. Sobrecarga

Mediante la *sobrecarga* podemos redefinir una función de manera que, manteniendo el mismo nombre, pueda recibir distinto número y tipos de parámetros.

Ejemplo:

Las siguientes funciones pueden coexistir sin conflictos bajo un mismo espacio de nombres.

```
#include <cmath>

double norma(double x) { return x < 0 ? -x : x; }

double norma(double v[], size_t n)
{
  using std::sqrt;
  double s = 0.0;

  for (size_t i = 0; i < n; ++i)
    s += v[i] * v[i];
  return sqrt(s);
}
```

Es el contexto el que determina cuál de las funciones se ejecutará:

```
const size_t n = 100;

double x;
double v[n];

cout << "|x| = " << norma(x) << endl
     << "|v| = " << norma(v, n) << endl;
```

En C++, las funciones pueden poseer parámetros con valores por omisión. Esto es, en el fondo, una forma de sobrecarga.

Ejemplo:

Se desea una función que calcule el logaritmo discreto (por defecto) de un número natural n en base b. La base será mayor que 1 y, por convenio, el logaritmo discreto de 0 será 0.

La función, que se llamará *log()*, tendrá el siguiente prototipo:

```
unsigned long log(unsigned long n, unsigned long b = 10);
```

Esto permitirá que durante una llamada a esta función se omita el parámetro b; en tal caso, se calculará el logaritmo discreto decimal, es decir, actuará cómo si hubiera recibido un 10 en b.

Por lo tanto, si la definición de la función es:

```
unsigned long log(unsigned long n, unsigned long b)
{
  unsigned long r = 0;

  while (n /= b)
    ++r;
  return r;
}
```

la declaración del parámetro por omisión equivale a definir la siguiente sobrecarga de la función:

```
unsigned long log(unsigned long n)
{
  unsigned long r = 0;

  while (n /= 10)
    ++r;
  return r;
}
```

Las plantillas, que veremos más adelante, también pueden entenderse como una forma de sobrecarga genérica.

También se pueden sobrecargar los operadores (con algunas excepciones), variando los tipos de los objetos con los que operan; lo que no se puede alterar es su sintaxis, ni precedencia, ni asociatividad. Esto resulta muy cómodo, sobre todo en la implementación de tipos de orientación matemática.

La sobrecarga también se extiende a las clases, que se presentarán a continuación: los constructores, funciones y operadores miembro de las clases pueden aparecer sobrecargados.

1.5.5. Clases

C++ introduce el concepto de clase permitiendo que una estructura (`struct`) contenga no sólo miembros de datos, sino también *funciones y operadores miembro*.

Por omisión, todos los miembros de una estructura son visibles desde el exterior (esto es necesario para mantener la compatibilidad con C). Para fomentar el buen uso del principio de ocultación de datos se introdujo la palabra reservada `class`, donde, por omisión, los miembros de una clase están ocultos al exterior.

Pero, para que el principio de ocultación de datos sea útil, es necesario poder especificar qué partes de una clase son visibles desde el exterior; esto llevó a la introducción de tres palabras reservadas: `public` (visible desde el exterior), `private` (oculto al exterior) y `protected` (se explicará en el capítulo 4).

Existen algunas funciones miembro especiales. De éstas, las más importantes son los *constructores*: se emplean para «construir» objetos, es decir, inicializar variables del tipo de la clase. Los constructores se distinguen por tener el mismo nombre que su clase y no especificar el tipo de devolución; implícitamente devuelven el objeto construido.

Ejemplo:

La clase *Reloj*, cuya declaración se presenta a continuación, representa un sencillo reloj digital. Consta de dos secciones: una pública y otra privada.

La sección pública contiene al constructor (cuyos parámetros son omitibles), dos funciones miembro modificadoras (alteran el objeto sobre el que actúan) y una función miembro observadora (se limita a observar su estado interno).

La sección privada contiene los miembros de datos: una variable entera para guardar el número de horas y otra para el de minutos.

reloj/reloj.h

```
1  // Clase Reloj
2
3  #ifndef RELOJ_H_
4  #define RELOJ_H_
5
6  class Reloj {
7  public:
8    Reloj(int h = 0, int m = 0);
9    void incrementarHoras();
10   void incrementarMinutos();
```

```
11    void mostrar();
12  private:
13    int horas,
14        minutos;
15  };
16
17  #endif
```

Para definir el constructor y las funciones miembro, hay que cualificar sus nombres. Al resolver el ámbito anteponiendo el nombre de la clase se evitan ambigüedades: queda claro que no se están definiendo funciones externas, sino miembros de la clase *Reloj*.

reloj/reloj.cpp

```
1   // Clase Reloj
2
3   #include <iostream>
4   #include <iomanip>              // para setfill() y setw()
5   #include "reloj.h"
6
7   // Constructor
8
9   Reloj::Reloj(int h, int m): horas(h), minutos(m) {}
10
11  // Funciones miembro modificadoras: incrementan los contadores
12
13  void Reloj::incrementarHoras()
14  {
15    horas = (horas + 1) % 24;     // circular
16  }
17
18  void Reloj::incrementarMinutos()
19  {
20    minutos = (minutos + 1) % 60; // circular
21  }
22
23  // Función miembro observadora: muestra las horas y los minutos
24
25  void Reloj::mostrar()
26  {
27    using namespace std;
28
29    cout << setfill('0')          // carácter de relleno: '0'
30         << setw(2) << horas      // horas: dos carácteres
31         << ':'                   // carácter separador: ':'
32         << setw(2) << minutos;   // minutos: dos carácteres
33  }
```

Nótese la sintaxis especial que emplea el constructor para «subconstruir» los valores de sus miembros de datos. En este caso no es obligatorio su uso, pero sí recomendable.

La función miembro *mostrar()* ilustra cómo se puede dar formato a la salida de datos mediante el empleo de *manipuladores*. Aquí aparecen dos de los más comunes: *setfill()* y *setw()*. Con ellos es fácil mostrar la hora del reloj en el formato *hh*:*mm*.

El programa de prueba construye un reloj, es decir, una variable del tipo *Reloj*. Inicialmente el reloj marca las 23:59. Tras incrementar, primero las horas y luego los minutos, marca las 00:00.

reloj/prueba.cpp

```
1  // Ejemplo de uso de la clase Reloj
2
3  #include <iostream>
4  #include "reloj.h"
5
6  int main()
7  {
8    using namespace std;
9    Reloj r(23, 59);
10
11   cout << "Reloj: "; r.mostrar();
12   cout << "\nIncrementamos las horas: ";
13   r.incrementarHoras(); r.mostrar();
14   cout << "\nIncrementamos los minutos: ";
15   r.incrementarMinutos(); r.mostrar();
16   cout << endl;
17 }
```

Se observa que la inicialización (construcción) de un objeto es análoga a una llamada a función. De hecho, se llama a una función, el constructor, y se le pasan los parámetros que aparecen entre paréntesis tras el nombre del objeto.

1.5.6. Plantillas

Las *plantillas* o *parametrizaciones de tipo* son un mecanismo de abstracción de la programación genérica que facilita la reutilización de código.

Mediante éste se permite abstraer una función de manera que en vez de trabajar con tipos específicos de datos, pueda trabajar con tipos cualesquiera, siempre que cumplan una serie de requisitos.

Como ejemplo básico, supongamos que queremos calcular el máximo de dos valores numéricos:

```
int max(int a, int b) { return a < b ? b : a; }
```

Esta función está ligada a un tipo concreto: recibe dos *int* y devuelve un *int*. No es válida por lo tanto para calcular el máximo de dos valores de tipo *long int* o *double*. Podría pensarse que basta definir esta función para el mayor tipo numérico que se desee emplear: las promociones y conversiones harían el resto.

No obstante, esta solución no es adecuada: las promociones y conversiones emplean tiempo, y las operaciones sobre el tipo mayor pueden ser más costosas que sobre el más pequeño.

Una solución, probablemente la empleada por un programador de C, sería definir una macro con parámetros:

```
#define MAX(a, b) ((a) < (b)) ? (b) : (a))
```

pero recuérdese que una macro *no* es una función y que tiene sus limitaciones y problemas (sobre todo, posibles efectos colaterales).

Por otro lado, crear una función para cada tipo con el que queramos trabajar es engorroso, aun empleando sobrecarga:

```
...
long int max(long int a, long int b) { return a < b ? b : a; }
float max(float a, float b) { return a < b ? b : a; }
double max(double a, double b) { return a < b ? b : a; }
...
```

y siempre queda la duda de si no aparecerá algún otro tipo para el que necesitemos de nuevo especializar la función.

Nótese que el código de todas estas funciones es el mismo: no existen diferencias entre las operaciones que hay que realizar con uno u otro tipo. Basta, eso sí, que para cada tipo esté definido el operador `<`, y que se puedan copiar valores, ya que el paso y la devolución se han especificado por valor. Por supuesto que, internamente, la comparación y la copia dependerán del tipo concreto y el código generado será distinto.

La solución es emplear una plantilla (*template*) que parametrice el tipo:

```
template <typename T> T max(T a, T b) { return a < b ? b : a; }
```

Se dice que T es un *parámetro de tipo* y que *max()* es, ahora, una *función paramétrica* o *genérica*. Nótese que se exige que los tipos de a y de b sean idénticos. La palabra reservada `typename` puede sustituirse, en este contexto, por `class`.

> La cabecera estándar `<algorithm>` contiene, entre otras, tres funciones paramétricas muy comunes: *min()*, *max()* y *swap()*. Éstas permiten, respectivamente, calcular el mínimo, el máximo e intercambiar valores de tipos arbitrarios.

Las plantillas deben ser especializadas por el compilador (o explícitamente por el programador) para cada tipo concreto con el que se emplee la función. Internamente, existirá una versión concreta de la función para cada tipo con el que se use. Por lo tanto, las plantillas son apropiadas para su colocación en ficheros de cabecera.

Conviene, siempre que se diseña una función paramétrica, pensar que los tipos paramétricos pueden ser arbitrariamente complejos y, siempre que sea posible, emplear paso y devolución por referencia. También puede ser útil sopesar si conviene que la función sea «en línea». En este caso, es posible escribir la función así:

```
template <typename T> inline
const T& max(const T& a, const T& b) { return a < b ? b : a; }
```

Un programa en el que se muestra su empleo es el siguiente:

max.cpp

```
1  #include <iostream>
2
3  template <typename T> inline
4  const T& max(const T& a, const T& b) { return a < b ? b : a; }
5
6  int main()
7  {
8    using namespace std;
9
10   {
11     cout << "Introduzca dos «int»:"
12             "\n\n";
13     cout << "a = "; int a; cin >> a;
14     cout << "b = "; int b; cin >> b;
15     cout << "max(" << a << ", " << b << ") = " << max(a, b)
16          << "\n\n";
```

```
17    }
18    {
19      cout << "Introduzca dos «double»:"
20              "\n\n";
21      cout << "a = "; double a; cin >> a;
22      cout << "b = "; double b; cin >> b;
23      cout << "max(" << a << ", " << b << ") = " << max(a, b)
24           << endl;
25    }
26  }
```

El compilador crea dos especializaciones de la función *max()*: una en la que T es *int*, y otra en la que es *double*. Los tipos concretos los deduce a partir de los de los parámetros reales.

Al igual que se pueden crear funciones paramétricas, se puede hacer lo propio con los operadores. Por ejemplo, supongamos que hemos definido los operadores `==` y `<` para un conjunto de tipos dado; el resto de los operadores relacionales puede definirse paramétricamente de la siguiente forma:

```
template <class T> inline
bool operator !=(const T& a, const T& b) { return !(a == b); }

template <class T> inline
bool operator <=(const T& a, const T& b)
 { return a < b || a == b; }

template <class T> inline
bool operator >(const T& a, const T& b) { return !(a <= b); }

template <class T> inline
bool operator >=(const T& a, const T& b) { return !(a < b); }
```

> En la cabecera estándar `<utility>` se encuentran definidos los operadores relacionales `!=`, `<=`, `>` y `>=` con la semántica habitual. Basta definir los operadores `==` y `<` para un tipo en cuestión e incluir la cabecera, para disponer automáticamente de los demás a través del espacio de nombres *std::rel_ops*.

Este mecanismo se extiende a los tipos, concretamente a las clases, para obtener lo que se denominan *clases paramétricas* o *genéricas*, es decir, plantillas de clases en las que se especifican parámetros de tipo.

Así, por ejemplo, en la biblioteca estándar de C++ aparecen definidos tres tipos complejos de coma flotante, uno para cada precisión: *float_complex*, *double_complex* y *long_double_complex*. Éstos son especializaciones de

una clase paramétrica, *complex*; la clase paramétrica y las versiones especializadas se encuentran en la cabecera `<complex>`. Esto es equivalente a lo siguiente:

```
typedef complex<float> float_complex;
typedef complex<double> double_complex;
typedef complex<long double> long_double_complex;
```

Igualmente, el tipo *string* es una especialización de la clase paramétrica *basic_string*: un *string* es equivalente a un *basic_string<char>*. Ambos, parametrización y especialización, se encuentran en `<string>`.

EJEMPLO:

La clase paramétrica *Par* permite representar pares de objetos de tipos arbitrarios.

par/par.h

```
1  // Clase para manejar pares de componentes arbitrarias
2
3  #ifndef PAR_H_
4  #define PAR_H_
5
6  template <typename T1, typename T2> class Par {
7  public:
8    Par(const T1& x = T1(), const T2& y = T2()): x(x), y(y) {}
9    T1& primero() { return x; }
10   T2& segundo() { return y; }
11   const T1& primero() const { return x; }
12   const T2& segundo() const { return y; }
13 private:
14   T1 x;
15   T2 y;
16 };
17
18 #endif
```

Nótese cómo el constructor posee como valores por omisión para sus parámetros los que crean los constructores por omisión de sus tipos respectivos.

En aras de la generalidad, cada una de las funciones miembro accesoras, *primero()* y *segundo()*, posee dos versiones: una para objetos no constantes y otra para objetos constantes.

El siguiente programa permite comprobar su uso:

par/prueba.cpp

```
1  #include <iostream>
2  #include <string>
3  #include "par.h"
4  using namespace std;
5
6  typedef Par<string, double> nombre_estatura;
7
8  void mostrar(const nombre_estatura& par);
9
10 int main()
11 {
12   nombre_estatura par;
13
14   cout << "¿Nombre?   "; cin >> par.primero();
15   cout << "¿Estatura? "; cin >> par.segundo();
16   mostrar(par);
17 }
18
19 void mostrar(const nombre_estatura& par)
20 {
21   cout << par.primero() << " (" << par.segundo() << "cm)" << endl;
22 }
```

El tipo *nombre_estatura* es simplemente un par formado por un *string* y un *double*. El objeto *par* es simplemente un ídem inicialmente constituido por la cadena vacía y el número 0,0.

Cuando *primero()* y *segundo()* se emplean en *main()*, lo que hacen es permitir la modificación del valor de *par* a través de la referencia que devuelven (son las versiones no constantes). Sin embargo, cuando se usan en *mostrar()* actúan sobre un objeto constante observando su contenido (son las versiones constantes).

1.5.7. Excepciones

Las *excepciones* son condiciones excepcionales que pueden surgir durante la ejecución del programa.

C++ posee un potente mecanismo de gestión de excepciones que permite:

1. Especificar las excepciones que puede provocar una función.
2. Provocar la aparición de una excepción.

3. Comprobar si ha aparecido una excepción.
4. Gestionarla en cualquier punto del programa desde donde se haya llamado, directa o indirectamente, a dicha función.

En la terminología de las excepciones, se dice que una excepción que se «lanza» (*throw*), se puede «recoger» (*catch*) posteriormente tras «intentar» (*try*) ejecutar un fragmento de código.

Ejemplo:

A continuación se define una clase *Nif* cuyo constructor está preparado para lanzar una excepción (en este caso, un objeto de tipo *char*).

nif/nif.h

```
1  // Clase para el NIF (número de identificación fiscal)
2
3  #ifndef NIF_H_
4  #define NIF_H_
5
6  typedef unsigned long int Dni;
7
8  class Nif {
9  public:
10   Nif(Dni, char) throw(char);
11   // ...
12 private:
13   Dni dni;
14   char letra;
15 };
16
17 #endif
```

Se especifica que el constructor únicamente puede lanzar dicha excepción mediante la lista `throw` que lo acompaña tanto en su declaración como en su definición. En caso de no aparecer una lista `throw` se entiende que puede lanzarse cualquier excepción.

nif/nif.cpp

```
1  // Clase para el NIF (número de identificación fiscal)
2
3  #include <cctype>
4  #include "nif.h"
```

```
5
6  // Constructor
7
8  Nif::Nif(Dni n, char c) throw(char)
9  {
10   using std::toupper;
11   char letra_correcta = "TRWAGMYFPDXBNJZSQVHLCKE"[n % 23];
12
13   if ((c = toupper(c)) != letra_correcta)
14     throw c;
15   dni = n;
16   letra = letra_correcta;
17 }
```

Dentro del cuerpo del constructor, se lanza la excepción usando la palabra reservada `throw` seguida de una expresión del tipo declarado. Esto se hace cuando se detecta la condición excepcional, es decir, cuando la letra suministrada no corresponde con el número de DNI.

En la función *main()* se leen de la entrada estándar un número de DNI y una letra. Con ambos se construye un objeto de la clase *Nif*.

nif/prueba.cpp

```
1  // Ejemplo de gestión de excepciones
2
3  #include <iostream>
4  #include "nif.h"
5
6  int main()
7  {
8    using namespace std;
9
10   try {
11     cout << "DNI:   ";
12     Dni dni;
13     cin >> dni;
14     cout << "Letra: ";
15     char letra;
16     cin >> letra;
17     Nif nif(dni, letra);
18     // ...
19     cout << "Todo en orden" << endl;
20   }
21   catch(char letra) {
22     cerr << "La letra «" << letra << "» es incorrecta.\a" << endl;
23   }
24 }
```

El bloque `try` que aparece en *main()* permite delimitar la zona de código donde va a ser comprobado el lanzamiento de la excepción. A un bloque `try` siguen uno o varios `catch` donde se recogen y se gestionan las posibles excepciones. En este caso, si se produce la excepción, el `catch` la recoge y la ejecución continúa tras él, con lo que se termina el programa.

Ejercicios

E1.1. Familiarícese con su entorno de trabajo: en particular, con el editor. Merece la pena invertir un poco de tiempo en aprender a manejar un buen editor. Practique, al menos, las operaciones generales de sangrado, búsqueda, sustitución, selección, corte, copia y pegado.

E1.2. Compruebe, mediante pequeños programas, las diferencias explicadas de C++ con respecto a C que hacen que un fragmento de código C sea incorrecto en C++.

E1.3. Escriba un programa que emplee una unión anónima global para mostrar la representación binaria de un número entero. Pruébelo con distintos números. ¿A qué conclusiones llega sobre la representación que emplea su sistema? ¿Se puede confiar en que esto sea así en todos los demás?

E1.4. Cree una función *raiz()*, capaz de calcular la raíz n-ésima discreta (por defecto) de un número natural. Si el exponente no se especifica, la función calculará la raíz cuadrada discreta.

Esta función, junto con la función *log()* ya vista, la incluirá en un espacio de nombres llamado *FuncionesDiscretas*; para ello, cree dos ficheros: `fd.h` y `fd.cpp`. En un módulo separado, `prueba.cpp`, escriba un programa de prueba sencillo. Por último, haga un *makefile* y compruebe que todo va bien.

E1.5. Modifique la clase *Reloj* para manejar también los segundos. ¿Es necesario retocar el código cliente para mantener su anterior comportamiento?

E1.6. Parametrice la función *intercambiar()* de manera que sea capaz de hacer lo propio con valores de un tipo arbitrario.

E1.7. Modifique la función *raiz()* para calcular la raíz de un entero, en vez de un natural. Si el entero es negativo, lo lanzará como excepción.

Cree un programa de prueba que, mientras se vayan introduciendo números, vaya calculando su raíz cuadrada discreta. Si se produce una excepción, el programa debe terminar con un mensaje indicativo.

Capítulo 2

Tipos, operadores y expresiones

2.1. Introducción

Se estudiarán aquí unos aspectos básicos tanto de este lenguaje como de muchos otros. Se repasará el concepto de variable, junto con ciertas variantes; luego el de tipo de datos, y se verán los que tiene el lenguaje incorporados y algunos otros muy importantes que están en la biblioteca estándar. A continuación se estudiarán los operadores y por último las expresiones.

Si se poseen ya conocimientos de programación, mucho material de este capítulo se puede saltar o leer rápidamente, y si además se sabe C, como se supone, aún más. Pero de todas formas conviene su lectura porque el C++ incorpora muchas mejoras y diferencias con el C.

2.2. Constantes, variables y referencias

Una *variable* no es más que un nombre para una zona de memoria en la que se pueden guardar datos de un cierto tipo (§2.3).

El nombre se conoce también como *identificador*, y debería ser lo más descriptivo posible de lo que se va a guardar.

> Un identificador se forma mediante las letras del alfabeto inglés, los dígitos del 0 al 9 y el subrayado, y no debe empezar por un dígito. Se diferencian las mayúsculas de las minúsculas. El subrayado como primer carácter no debería emplearse, pues se reserva para nombres de la biblioteca estándar. El Estándar admite también caracteres Unicode,

como ocurre ya en Java, así que ya se podría poner *año* en vez de *ano*, por ejemplo. El compilador traduce internamente los caracteres Unicode a la forma `\u`*yyyy* o `\u`*xxxxyyyy* (la primera es una abreviatura de `\u0000`*yyyy*), siendo *x* e *y* dígitos hexadecimales.

En cuanto al máximo número de caracteres significativos que puede tener un identificador, C++ no impone ningún límite, pero puede que el sistema sí, sobre todo porque algunas cosas no están al alcance del compilador, como los identificadores globales o externos, que pertenecen al ámbito del enlazador. En este caso se garantizan 6 caracteres únicos como mínimo.

2.2.1. Declaración, definición e inicialización

Las variables deben *declararse* antes de su uso, y para ello se escribe el tipo, el identificador y, opcionalmente, una *inicialización*; por último, un punto y coma. Si hay varias del mismo tipo, pueden separarse por comas, dando el tipo solamente al principio.

La inicialización consta del signo igual seguido de un valor del tipo apropiado, normalmente una constante *literal*. Un literal no es más que un dato o valor pero sin identificador asociado, como `4`, `6.23e23`, `'A'` o `"Agua"`; si el compilador le asigna o no un espacio de almacenamiento, es cosa suya y de las circunstancias. En C++ el inicializador puede ir también entre paréntesis (para los tipos básicos) siguiendo la sintaxis de una función especial llamada *constructor* que se estudiará en el capítulo sobre las clases.

Una *definición* implica que el compilador reservará un espacio de memoria para el almacenamiento de una variable, quizá poniendo algún valor inicial en él, o que generará código en el caso de una función. Normalmente una definición sirve también de declaración. La declaración sirve simplemente de información para el compilador. En el caso de las funciones, la diferencia entre declaración y definición está bien clara. En el caso de las variables, una declaración solamente, sin definición, se consigue anteponiendo la palabra reservada `extern`, lo que le indica al compilador que la definición está en otro sitio del programa, normalmente en otro módulo o fichero. Una inicialización implica definición.

Ejemplo:

En la última línea se muestran dos estilos que hay para construir identificadores formados por varias palabras: separarlas por subrayados, o poner las iniciales en mayúscula sin otra separación (lo que se conoce como «notación húngara»).

```
extern int i;              // sólo declaración
double interes = 0.47;     // con inicialización
int j(5);                  // otro estilo de inicialización
char caracter_anterior, CaracterAnterior = 'a', c;
```

El lenguaje C++, como el C, distingue mayúsculas de minúsculas, y sobre cuándo emplear un tipo de letra u otro hay muchas discusiones. Lo mejor quizás sea adoptar un estilo y ser siempre coherente con él. Lo único que parece universalmente aceptado es emplear todas las letras mayúsculas para las macros del preprocesador[1].

Es muy común también emplear todas las letras minúsculas para variables locales, especialmente automáticas.

2.2.2. Especificación de almacenamiento

Cuando se define un objeto (variable o función), se tiene la opción de especificar su duración, en qué zona de memoria se reservará espacio para guardarlo, y cómo el compilador tratará los datos que aloje.

Variables globales

Son las que se definen fuera de los cuerpos de las funciones y están disponibles en todas las partes del programa, incluso en otros ficheros. Su duración es la del mismo programa, hasta que acaba su ejecución.

Una variable global debe definirse en un fichero y declararse en los demás donde se pretenda utilizar. A menudo su declaración se escribe en un fichero de cabecera para que pueda ser compartida por los demás módulos. La declaración se consigue con la palabra reservada `extern`.

Ejemplo:

Esto es una demostración de cómo se utilizaría una variable global. Se tiene un programa compuesto por dos ficheros. En el primero se define la variable *global*, y para poder utilizarla en el segundo, se declara.

[1] Y sin embargo, una macro estándar, precisamente *__cplusplus* , está en minúsculas. Esta macro está definida (y vale 1) si se está preprocesando, obviamente, código fuente en C++.

global.cpp

```
1  // Demostración de variable global
2
3  int global;      // definición. global == 0
4
5  extern void f(); // declaración. extern es opcional
6
7  int main()
8  {
9    f();          // global == 47
10
11   return global;
12 }
13
```

global2.cpp

```
1  // Acceso a una variable global
2
3  extern int global; // declaración
4  // (el enlazador resuelve la referencia)
5
6  void f()
7  {
8    global = 47;    // ya se puede emplear aquí
9  }
```

El compilador reserva memoria para almacenar la variable al ver la definición en `global.cpp`. Como el fichero `global2.cpp` se compila por separado, hay que informar al compilador de la existencia de *global* antes de su empleo mediante la declaración. Cuando se enlazan los dos módulos objeto obtenidos de la compilación, el enlazador resuelve la referencia a *global* en *f()*.

En C++, todas las funciones son globales; por tanto, a la hora de declarar una función, la palabra `extern` es redundante y no se suele poner.

Las variables globales, por regla general, deben evitarse en lo posible. Si no se inicializan, el compilador las inicializa a 0.

Variables locales

Las variables locales residen dentro de un ámbito; son *locales* a una función o bloque. Por tanto sólo son *visibles* dentro del bloque donde se definan.

Si una variable local *oculta* a una global por tener el mismo nombre (se dice entonces que la global no es *visible*), en C++ es aún posible referirse a la variable global mediante el operador de resolución de ámbito global `::`.

Ejemplo:

```
int a = 1;    // global
void f(int a) // local
{
  if (a == ::a) // ::a es la «a» global
    return;
  else
    ::a = a + 1;
}
```

Por supuesto, hay que evitar estas situaciones.

Variables automáticas

Son variables locales cuya vida o duración va desde el principio del bloque donde se han definido hasta que el control de flujo del programa llega al fin de dicho bloque. En C++ no hace falta definir una variable justo al principio de un bloque, como en C; sin embargo el compilador reserva la memoria al entrar en el bloque, si bien aún no puede utilizarse el identificador correspondiente. Se puede hacer explícito que una variable sea automática mediante la palabra reservada `auto`, pero es redundante, pues a falta de otra palabra, una variable local siempre es `auto`mática.

Ejemplo:

```
void f(int a) // 'a' es local a f() y auto
{ // se reserva espacio para 'a', ¡y para 'd' y para 'b'!
  auto double d;  // 'auto' es redundante
  d = 3.5;
  // ...
  int b = 0;  // a partir de aquí se puede usar 'b'
  // ...
  { // Otro bloque: se crea 'c'
    char c;
    // ...
  } // se destruye 'c'
} // se destruyen 'a', 'd' y 'b'
```

Las variables automáticas no se inicializan solas; si no lo hace el programador explícitamente, contendrán *basura*: un valor cualquiera, lo que hubiera en la memoria.

Las variables automáticas se almacenan en la *pila* de la función donde se definen.

Variables registro

Una variable *registro* es un tipo de variable local automática. La palabra reservada `register` le dice al compilador: «intenta que el acceso a esta variable sea lo más rápido posible». Cómo lo haga, depende del sistema, pero el nombre sugiere que a menudo se intenta poner la variable en un registro del procesador. No hay garantía ninguna de que esto sea así, ni de que la velocidad de acceso aumente; sólo es una sugerencia para el compilador. Sin embargo, no puede tomarse la dirección de una variable `register`, y sólo algunos tipos pueden ser `register`: enteros, caracteres, punteros.

Ejemplo:

El empleo de `register` debería dejarse sólo para variables, por ejemplo de control de bucles `for`, a las que se va a acceder muy frecuentemente dentro de una función. Sin embargo no suele utilizarse casi nunca, pues la optimización del acceso a las variables es tarea del compilador. La mayoría de compiladores de hoy día se encarga automáticamente o con alguna opción de optimización de poner en registros las variables que estime oportuno.

```
void f(register int a) // un parámetro formal puede ser register
{
  for (register int i = 0; i < 10000; i++)
    a += i;
  // ...
}
```

Variables estáticas

La palabra reservada `static` tiene varios significados distintos. Ya se ha visto que las variables automáticas desaparecen o se destruyen al final del bloque donde se han definido. Cuando se llama de nuevo a la función, se vuelve a reservar espacio en la pila de la función y se vuelve a inicializar, si es el caso. Si se quiere en cambio que la duración de la variable sea hasta

el final del programa, se puede definir *estática* y darle un valor inicial (si no, se inicializa a 0, como las globales). La inicialización sólo tiene lugar la primera vez, cuando se crea la variable, y ésta retiene el valor que tuviera entre llamadas a la función donde está. De esta manera, una función puede «recordar» información local entre llamadas.

La gracia de las variables locales estáticas es que precisamente son locales, su ámbito está restringido a la función o bloque donde se definan; esto ayuda a localizar posibles errores.

Ejemplo:

```
1  // Ejemplo de dato estático local
2
3  #include <iostream>
4
5  int f()
6  {
7    static int c = 0; // la inicialización a 0 es redundante
8    return ++c;
9  }
10
11 int main()
12 {
13   using namespace std;
14   for (int i = 0; i < 10; i++)
15     cout << f() << endl;
16 }
```

Al ejecutar el programa, se ve que cada vez que se llama a la función *f()* se imprime un valor diferente. Sin la palabra `static`, siempre se imprimiría el valor 1.

El segundo significado de la palabra `static` se obtiene al aplicarlo sobre una variable global o sobre una función. En este caso quiere decir: «este nombre no está accesible fuera de este fichero». Se dice que el nombre afectado tiene alcance de fichero.

> Este empleo de `static` se mantiene por compatibilidad con C, pero en C++ está desaconsejado en favor de los espacios de nombres anónimos.

EJEMPLO:

Como demostración, compílese el siguiente programa, separado en dos ficheros. No habrá ningún problema al compilar cada uno, pero al enlazar los módulos se producirá un error.

fstatic1.cpp

```
1  // Demostración de alcance de fichero.
2  // Al enlazar este módulo con fstatic2.o
3  // se producirá un error de enlazado.
4
5  static int af; // alcance de fichero: sólo accesible aquí
6
7  int main()
8  {
9    af = 1;
10 }
```

fstatic2.cpp

```
1  // Intento de referenciar 'af', que está en el
2  // módulo fstatic1.o
3
4  extern int af;  // declaración
5
6  void f()
7  {
8    af = 100;
9  }
```

Al enlazar:

```
% c++ fstatic1.o fstatic2.o
fstatic2.o: In function 'f(void)':
fstatic2.o(.text+0x5): undefined reference to 'af'
collect2: ld returned 1 exit status
```

El enlazador no ha podido encontrar la variable global *af*, aunque la hemos declarado, pues fue definida como `static` en el otro fichero; si la hubiéramos definido en `fstatic2.cpp` (sin `extern`), hubiera sido una variable distinta de la otra, aunque con el mismo nombre.

La palabra `static` también se puede emplear dentro de una *clase*, pero eso se verá en el capítulo correspondiente.

Enlace

El enlace describe cómo ve el enlazador el almacenamiento creado en memoria para representar un identificador. Un identificador se representa mediante un almacenamiento en memoria para contener un dato o una función compilada. Hay dos tipos de enlace: interno y externo.

Enlace interno Significa que el almacenamiento que se reserva para representar el identificador es solamente para el fichero que se está compilando. El enlazador ve el símbolo pero sabe que sólo sirve para ese fichero; por lo tanto se puede emplear el mismo identificador en otro fichero sin que haya ningún conflicto (aunque no suele ser buena práctica, por supuesto). El enlace interno se especifica mediante la palabra reservada `static`, o mejor con los espacios de nombres anónimos.

Enlace externo Significa que el almacenamiento que se reserva para un identificador es único en el programa, lo representa para todos los ficheros que lo contengan. El almacenamiento se reserva una vez, y el enlazador debe resolver todas las otras referencias a este almacenamiento. Las variables globales[2] y las funciones tienen enlace externo, a menos que se emplee `static`, y para acceder a ellas desde otros módulos hay que declararlas con la palabra `extern`, opcional en el caso de las funciones.

Sin enlace Las variables locales automáticas existen sólo temporalmente, en la pila, mientras una función se está ejecutando. El enlazador no sabe nada acerca de ellas, y no tienen pues enlace.

2.2.3. Constantes

Muchas veces se necesita un número o dato que no va a variar durante la vida del programa. Escribir el literal muchas veces puede llevar a errores, sobre todo si se decide cambiarlo, y a la hora de leer el programa hay que acordarse de su significado. La solución que se empleó y aún se emplea en C hace uso del preprocesador y su capacidad de definir *macros*:

```
#define LIMITE 255
```

Esto hace que a partir de ese punto y hasta el final del fichero, donde aparezca en el texto del programa (salvo dentro de comentarios o cadenas de

[2]Excepto las constantes en C++, como se ve a continuación en §2.2.3.

caracteres literales, evidentemente) el lexema (unidad sintáctica) *LIMITE* será sustituido por el preprocesador por el texto de reemplazo, que en este caso es 255.

Ejemplo:

```
int vec[LIMITE], i;
/* ... */
for (i = 0; i < LIMITE; i++)
  vec[i] = i * i;
```

Como se ve, se mejora la legibilidad del programa. Las definiciones de macros se suelen poner en ficheros de cabecera, para que puedan ser compartidas por más de un fichero fuente.

Veamos las ventajas e inconvenientes de las macros, y la solución opcional que nos ofrece el C++.

Las ventajas de las macros son:

- Mejoran la legibilidad del programa, y si hay que cambiar su valor, se hace en un solo sitio.
- Es posible anular su definición en cualquier punto, mediante la directiva del preprocesador `#undef` .
- Es posible definirlas sin tener que modificar ningún fichero, simplemente recompilando con la opción `-D` del compilador (u otra similar, consulte el manual de su compilador).
- Permiten la compilación condicional, que es algo muy importante.

Y las desventajas son:

- Al ser tratadas por el preprocesador, el compilador no sabe nada de ellas: sólo ve el texto de reemplazo. Por lo tanto las macros no obedecen ninguna regla de alcance, visibilidad o ámbito.
- Debido a lo anterior, un depurador o trazador de código tampoco las reconoce, lo que puede llegar a ser bastante incómodo.
- Además, los parámetros se evalúan tantas veces como aparezcan en la expresión, y conviene encerrarlos siempre entre paréntesis en el texto de reemplazo porque los operadores pueden mezclarse inconvenientemente. Todo esto puede dar lugar a efectos no deseados, llamados *efectos colaterales*.

- Las macros no tienen información de tipo. Esto puede llegar a ocultar fallos muy difíciles de detectar.

Para evitar estos inconvenientes, en C++ se introdujo la palabra reservada `const`, abreviatura de *constant*, o sea, constante.

> Al poco de su introducción en C++, el comité ANSI que estaba normalizando el lenguaje C la tomó prestada y la introdujo también en C, pero en C++, cuyo estándar es mucho más reciente, siguió evolucionando, de forma que en C aún es preciso el empleo de las macros muchas veces, no así en C++.

La palabra `const` trae la sustitución del valor al dominio del compilador, no del preprocesador, de forma que si ahora definimos:

```
const int limite = 255;
```

podemos emplear *limite* en cualquier sitio donde el compilador pueda conocer su valor en tiempo de compilación. El compilador de C++ evitará si puede el obtener almacenamiento para *limite*, cosa en que se diferencia de C, donde *limite* no sería más que una variable (almacenada en memoria por tanto como otra cualquiera) pero exclusivamente de lectura. Por eso en C++ se puede hacer:

```
int vec[limite];
```

mientras que en C no, pues el compilador de C no puede saber el valor de *limite* en tiempo de compilación ya que ocupa un lugar en la memoria, está almacenado en algún sitio. En C++ el compilador trata a un `const` como a una macro: en tiempo de compilación sustituye su valor donde se utilice el identificador. Si puede evitarlo, ni siquiera reserva un espacio en memoria para su almacenamiento, aunque esto no es posible si se toma la dirección del `const` o si el tipo es muy complicado.

Para poder sustituir efectivamente a las macros del preprocesador, se debería poder definir `const`s en ficheros de cabecera igual que con aquéllas. En C++ esto es posible sin problemas debido a que un `const` en C++ (pero no en C) tiene *enlace interno*: esto es, a menos que se declare explícitamente `extern`, sólo es visible en el fichero en el que se defina; si lo ponemos en una cabecera, será visible en los ficheros en los que se incluya ésta. Al tener enlace interno, el enlazador no los *ve*, por lo que no hay problemas de definiciones múltiples.

Un objeto `const` siempre debe inicializarse en C++ (en C, si no se inicializaba explícitamente y era global o estático, se inicializaba automáticamente

con sus bits a 0). Esto es lógico, pues luego no se le puede asignar nada. Recuérdese que una inicialización es algo distinto de una asignación: una inicialización sólo ocurre durante la creación o definición de una variable u objeto.

Es posible mezclar macros y constantes para obtener las ventajas de ambas.

EJEMPLO:

```
#include <math.h>

const double pi =
#ifdef M_PI
  M_PI
#else
  3.14159265358
#endif
  ;
```

En muchos sistemas se define el valor del número π como la macro *M_PI* en el fichero de cabecera `<math.h>`, pero esto no es estándar, así que hemos definido la constante *pi* como el valor de dicha macro si está definida, y si no, como el valor de π con 11 decimales.

Esta técnica de compilación condicional, nos permite poder cambiar el valor de una constante sin modificar el fichero, sólo recompilando con la opción `-D`:

```
#ifndef LIMITE
#define LIMITE 255
#endif

const int limite = LIMITE;
```

Si compilamos normalmente, *LIMITE* no estará definido y por tanto se define con el valor 255, que toma *limite*. Pero si compilamos con

```
c++ -DLIMITE=1024 ...
```

(o lo equivalente en el compilador que se utilice, habría que consultar su manual), entonces *LIMITE* ya tendrá un valor, se salta la línea `#define` y *limite* toma el valor dado por nosotros al compilar, 1024.

Las constantes no sólo se han inventado para sustituir a las macros del preprocesador, sino que deben emplearse cada vez que tengamos un objeto,

inicializado con un valor que puede estar producido en tiempo de ejecución o de compilación, que sepamos que ya no debe cambiar. Esto nos ayuda a detectar errores si accidentalmente lo modificamos. Además ayuda al compilador a producir un mejor código.

Cuando el modificador `const` se aplica a punteros, la cosa se complica un poco.

Ejemplo:

```
int i = 5, j = 0;
const int* p;       // p es un puntero a const int
int* const pp = &i; // pp es un puntero const a un int
int const* q;       // q es un puntero a un int que es const
const int* const qq = &i; // 0: int const* const qq = &i;

p = q = &i; // p y q no tuvieron que inicializarse
*p = 6;     // ERROR, *p es const (¡aunque no i!)
*pp = 6;    // Bien, pp es const, pero no a donde apunta
p = &j;     // Bien, p no es const
*q = 6;     // ERROR, *q es const
q = &j;     // Bien, q no es const
qq = &j;    // ERROR, tanto qq como
*qq = 7;    //        *qq son const
```

Para acordarse, se puede pensar en que `*const` es un «puntero constante», o sea, que debe inicializarse con una dirección, a la cual no puede ya dejar de apuntar; mientras que si el asterisco no está inmediatamente a su izquierda, lo que es constante es el objeto al que apunte el puntero, pero sólo a través de éste; es decir, no se puede modificar el objeto a través del puntero. Esto es muy útil principalmente en funciones que reciben punteros como parámetros, para evitar que accidentalmente puedan modificar el objeto apuntado, como en

```
unsigned int Strlen(const char* s)
{
  const char* p = s;
  for(;;)
    if (*s++ == '\0') break;
  return (unsigned int)(--s - p);
}
```

Esto es una mala implementación de la función de la biblioteca estándar de C *strlen()*, que devuelve el número de caracteres de una cadena de caracteres de bajo nivel (§2.3.2); evidentemente se puede hacer mejor (y mejor no hacerla, pues la de la biblioteca será aún mejor), pero ilustra el hecho de que si por

error hubiéramos puesto un solo `=` en vez de dos en el `if`, el compilador nos daría un error, pues estaríamos modificando $*s$, lo que se detectaría debido al `const` del parámetro.

Muchas más cosas se pueden decir de las constantes, pero aún no se tienen los elementos suficientes; se trata de su empleo dentro de las clases.

2.2.4. Volátiles

La sintaxis de variables *volátiles* es idéntica a la de las constantes, pero el significado no. La palabra reservada `volatile` (*volátil*) significa: «este dato puede cambiar más allá del conocimiento del compilador»; o sea, desde un suceso externo al programa, quizá debido a un proceso concurrente que lo altere, o a que la variable esté ligada, por ejemplo, a una zona de memoria modificada por un dispositivo *hardware* de comunicaciones, como un puerto de entrada/salida.

La información proporcionada al compilador por la palabra `volatile` se resume en: «no optimices el acceso a esta variable». Por ejemplo, si el compilador *dice*: «he guardado ese dato antes en un registro, y no lo he tocado», normalmente no tendría que volver a leerlo de nuevo. Pero si el dato es *volátil*, el compilador no puede hacer esa suposición, porque el dato puede haber sido cambiado externamente, y debe releerlo en vez de optimizar el código.

La palabra `volatile` no es lo contrario de `const`; de hecho, es perfectamente posible tener variables `const volatile` (el orden da igual) que no se puedan modificar directamente en el programa, pero que sí puedan modificarse por un suceso externo a él.

2.2.5. Referencias

Las referencias son nuevas en C++, no existen en C. Se puede pensar en ellas como una facilidad o comodidad sintáctica, pues no son más que punteros constantes desreferenciados automáticamente por el compilador, con lo que nos ahorramos el operador `*`, y trabajar con direcciones (`&`); sin embargo, son imprescindibles para poder realizar el *constructor de copia*, que se verá en el capítulo 4.

El principal y prácticamente único empleo de las referencias es en el paso de parámetros a funciones, y en la devolución de sus resultados. Cuando definimos una referencia, el compilador no reserva memoria para almacenar nada, no crea almacenamiento; una referencia no es más que otro nombre, una especie de *alias* o sinónimo, para un objeto ya existente. Una referencia

debe pues inicializarse, y no podrá ya referenciar a otro objeto; además no hay forma de hacerlo, pues al intentar modificar su valor modificamos en realidad el del objeto referenciado. Una referencia no puede valer *NULL* , como los punteros: siempre referencian a un objeto válido.

Ejemplo:

```
int cola[longitud];
int& primero = cola[0];
int i = 5;
int& ri = i;
i = 6;
ri = 7;
const int& ref = 12;
```

Como se ve, para definir una referencia se pone entre el tipo y el identificador el signo &; aquí no es el operador de dirección, aunque se ha escogido este signo para dar a entender que una referencia no es más que un puntero «automático». Se ha inicializado la referencia *ri* con *i*, de forma que ya tanto *i* como *ri* se refieren al mismo sitio de memoria. Por eso, al final del ejemplo, *i* acaba valiendo 7.

En la última línea, ¿a qué posición de memoria hace referencia *ref*? El compilador debe reservar un espacio de memoria anónimo temporal para guardar el valor 12 y asocia a *ref* con ese espacio.

Como se ha dicho, prácticamente nunca se emplean las referencias directamente, sino sólo en el paso de parámetros y devolución de funciones.

Cuando un parámetro formal de función es una referencia, cualquier modificación de ésta *dentro* de la función afecta al parámetro real pasado a la función desde *fuera*. Lo mismo puede hacerse con punteros, pero la sintaxis es mucho más cómoda y clara.

Si una función devuelve una referencia, hay que tener el mismo cuidado que cuando devuelve un puntero: lo que referencie no debe destruirse al acabar la función, como ocurre por ejemplo con un objeto local `auto`.

Ejemplo:

El siguiente programa ilustra el paso de parámetros y la devolución de resultados por referencia y muestra las diferencias con el empleo de punteros.

referencias.cpp

```
1  int* f(int* x)
2  {
3    (*x)++;
4    return x; // Bien: x está fuera de este ámbito
5  }
6
7  int& g(int& x)
8  {
9    x++;       // el mismo efecto que en f()
10   return x;  // Bien: fuera de este ámbito
11 }
12
13 int& h()
14 {
15   auto int q;  // 'auto' no hace falta
16   // return q; // esto estaría mal
17   static int x;
18   return x;    // ahora sí: x vive tras este ámbito
19 }
20
21 #include <iostream>
22 using namespace std;
23
24 int main()
25 {
26   int a = 0, b, *p;
27   p = f(&a);  // con punteros: feo pero explícito
28   b = *f(&a); // más feo aún.
29   b = g(a);   // más cómodo y bonito, pero oculto el cambio
30   h() = ++b;  // x de h() vale 3
31   // comprobación de todo
32   cout << "a = " << a << "; &a = " << &a << endl
33        << "b = " << b << ";  p = " <<  p << endl
34        << "h() = " << h() << endl;
35 }
```

Dentro de *f()*, la dirección de a se copia en el puntero x, luego x apunta a a; al incrementar $*x$ estamos incrementando (modificando) efectivamente a, que pasa a valer 1. La función devuelve luego el valor de x, que es la dirección de a, que se copia en el puntero p. Observe que todo el paso ha sido por valor, pero se han pasado, o sea copiado, direcciones.

Dentro de *g()*, la referencia x se asocia a a, por lo que ahí x y a son lo mismo. Al incrementar x, a pasa a valer 2. Se devuelve la referencia x, que es lo mismo que a, por lo que b vale a, o sea 2.

Ahora se incrementa b y su valor incrementado, o sea 3, se asigna ¡a la función *h()*! Esto es posible porque *h()* devuelve una referencia no `const`. En realidad, se está almacenando el 3 en la variable estática x definida dentro de *h()*, actuando así esta función como una pseudo-variable.

La variable q no nos sirve porque es automática y se destruye al acabar *h()*, con lo que la función devolvería la referencia de un objeto destruido.

En la última instrucción se comprueba cómo p apunta a a; los valores finales de a y b, y el valor que devuelve *h()*, que no es otro que el de su variable interna x, que como es `static` no se destruye hasta que no termina el programa.

Como se ve, es más cómodo trabajar con referencias.

> Sin embargo hay autores que opinan que, al leer el código, no queda claro si la función va a modificar el parámetro real o no; por tanto ellos recomiendan que si una función va a modificar el parámetro que recibe, se empleen punteros, como en la función *f()* del ejemplo anterior, o como en C.
>
> Nosotros no estamos de acuerdo porque la mayoría de los lenguajes modernos tienen el paso por referencia, que se emplea además exhaustivamente en la propia biblioteca estándar de C++, y porque si la función va a modificar sus parámetros reales o no, es algo que debería quedar claro en el propio nombre de la función o en su documentación, en vez de tener que oscurecer el código con los punteros innecesariamente.

Además de para que una función pueda modificar sus parámetros, hay otro motivo para emplear referencias.

Se sabe que no conviene pasar estructuras grandes a funciones por valor, pues eso implica copiar muchos datos en cada llamada; la alternativa era en C pasar punteros a esas estructuras, quizá con la palabra `const` delante si en la función no debieran modificarse. Esto obligaba a hacer uso de los operadores `*` y `->` en vez de simplemente el operador punto de selección de miembro. Ahora se pueden sencillamente pasar referencias, y si no deben modificarse, referencias constantes.

Esto será especialmente útil en C++ cuando veamos las clases, que son como estructuras muy perfeccionadas.

Un empleo obligado de referencias constantes puede verse a continuación.

EJEMPLO:

pasconst.cpp

```
1  // Paso de referencias como constantes
2
3  void f(int&) {}
4  void g(const int&) {}
5
6  int main()
7  {
8    // f(1); // ERROR
9    g(1);
10 }
```

La llamada a *f()* produciría un error porque el compilador debe crear una referencia: lo hace reservando espacio para un *int*, inicializándolo a 1 y produciendo la dirección para enlazarla a la referencia. Ese almacenamiento *debe* ser `const` porque modificarlo no tendría sentido (1 siempre es 1, y además no tiene nombre fuera de *f()* con que referenciarlo).

A veces se necesita modificar el *contenido de un puntero* (la dirección que almacena) en vez del objeto al que apunte; en C, la declaración de una función que hiciera tal cosa sería

```
void f(int**);
```

y habría que llamarla pasándole la dirección del puntero, como en

```
int i = 47;
int* pi = &i;
f(&pi);
```

Con las referencias de C++, la sintaxis es mucho más clara; el parámetro de la función sería ahora una referencia a un puntero, y ya no hay que tomar la dirección del puntero.

EJEMPLO:

refptr.cpp

```
1  // Referencia a puntero
2
3  #include <iostream>
```

```
4  using namespace std;
5
6  void incrementar(int*& i) { i++; }
7
8  int main()
9  {
10   int* i = 0;
11   cout << "i = " << i << endl;
12   incrementar(i);
13   cout << "i = " << i << endl;
14 }
```

El resultado de ejecutar el programa es:

```
i = (nil)
i = 0x4
```

Lo que prueba que el puntero i ha cambiado (y de paso, si se fija, prueba que el tamaño de un *int* en el computador donde se ha probado el programa es de 4 bytes, puesto que el incremento del puntero ha sido justamente de 4).

2.3. Tipos de datos

2.3.1. Conceptos previos

Se estudiarán los tipos incorporados en el lenguaje, y un par de ellos muy útiles de la biblioteca estándar; antes, se definirá qué es un tipo de datos y se clasificarán.

Definición

Un tipo de datos determina cómo se interpretarán los bits almacenados en ciertas posiciones de memoria, y cuántos habrá que considerar. También determina las operaciones que se podrán realizar sobre los datos.

Una variable consta de un nombre y un tipo; una constante literal, o simplemente *literal*, sólo consta de un tipo, no tiene nombre.

Clasificación

Ciertos tipos están *incorporados* en el lenguaje, de forma que el compilador los conoce: sabe qué literales son de esos tipos, sabe cuánto ocupan, cómo interpretar sus bits, qué operaciones admiten y qué operadores pueden actuar sobre ellos, sabe si se pueden convertir de unos a otros, y cómo, etc.

Algunos son *tipos básicos* del lenguaje: booleanos o lógicos, caracteres, números enteros y reales en coma flotante.

Otros se construyen a partir de éstos, combinándolos de alguna manera, por lo que hay una infinita gama de ellos, si bien sólo hay algunas formas de componerlos: son los tipos *agregados*, como estructuras, uniones, enumeraciones, vectores y cadenas; podríamos meter aquí punteros, referencias y funciones.

Un tipo especial de agregado existente en C++ es la *clase* (una forma de estructura), que se estudiará en detalle en el capítulo 4. Mediante ella el programador puede definirse sus propios tipos de datos, con sus operaciones y operadores, conversiones, y formas de inicialización y destrucción. Éstos son pues *tipos definidos por el usuario*, y suelen ser implementaciones de *tipos abstractos de datos* (TAD, o ADT en inglés).

En este capítulo se verán brevemente los tipos incorporados en el lenguaje. Como son prácticamente los mismos que en C, se describirán muy brevemente recalcando sólo las diferencias. A continuación se estudiarán de modo muy sucinto dos tipos definidos en la biblioteca estándar de C++, que corresponden a cadenas de caracteres y vectores y extienden enormemente la escasa funcionalidad que tienen los tipos correspondientes incorporados en el lenguaje.

2.3.2. Tipos de datos incorporados

Lógico

El tipo lógico o *booleano*[3] no existe en C; en C cualquier expresión entera distinta de 0 se considera «cierta», y la expresión que da 0 se considera «falsa». Como la mayoría de los programadores acababan definiendo un tipo lógico a base de macros, `typedef`s o enumeraciones, y dándole cada uno un nombre distinto, el comité de normalización del C++ decidió crear un tipo lógico, y que la mejor forma era incorporarlo en el lenguaje. El tipo lógico se denota con la palabra reservada `bool`, abreviatura de *boolean*, y las variables de este tipo sólo pueden tomar los valores lógicos `true` (*verdadero*) o `false`

[3] En honor al lógico y matemático británico George Boole (Lincoln, 1815–Ballintemple, cerca de Cork, 1864).

(*falso*), que son también palabras reservadas, y los únicos literales de este tipo que hay.

Para mantener la compatibilidad con el C y con el C++ anterior, la conversión entre enteros y booleanos está perfectamente definida: el 0 se transforma en `false` y cualquier otro valor en `true`; `true` se convierte en 1 y `false` en 0. Esto permite mantener intacto todo el código anterior que no utilice el tipo *bool*.

Otros cambios en el lenguaje implican a los operadores lógicos y relacionales, al operador condicional y a las estructuras de control condicionales y de iteración: donde antes esperaban o devolvían una expresión entera, ahora esperan o devuelven un *bool*.

> El incremento o decremento de una variable booleana, práctica extendida en C para cambiar de falso a verdadero una variable entera, es posible pero desaconsejado, pues implica una conversión de *bool* a *int* y de nuevo de *int* a *bool*, aparte de oscurecer el código. Lo mejor es una asignación directa a `false` o a `true`.

Caracteres

Pensados originalmente en C para almacenar caracteres, y de ahí el nombre, son en realidad tipos de números enteros pero de tamaño 1 byte. Vienen en tres formas: *char*, *signed char* y *unsigned char*. Las dos últimas se emplean cuando se quieren guardar números de 1 byte y especificar el signo, puesto que en la primera forma no está definido si el número almacenado es con signo o sin él. La primera forma es totalmente válida para guardar caracteres sin más.

Estos tipos se emplean sobre todo en agregados: punteros y vectores (cadenas de caracteres).

Los literales de este tipo se forman escribiendo un carácter entre apóstrofos o comillas simples, como `'A'`; el valor de este literal es el código numérico correspondiente a la letra *A* en el código de caracteres de la máquina; como existen muchos códigos, este valor puede variar con el sistema. El código más apropiado por ahora para España es el ISO-8859–1, también conocido como ISO-Latin1; uno nuevo, el ISO-8859–15 o ISO-Latin9, es también apropiado e incorpora el signo del euro, €. Cuando un carácter no es imprimible, como el salto de página o de línea, o no es posible ponerlo de esta forma, como el mismo apóstrofo, se pueden utilizar las *secuencias de escape*, que constan de la barra invertida y un carácter que toma un significado especial. Las secuencias de escape estándar son las de la tabla 2.1.

`\a`	Alerta (pitido)	`\\`	Barra invertida
`\b`	Espacio atrás	`\?`	Interrogación derecha
`\f`	Salto de página	`\'`	Comilla simple
`\n`	Nueva línea	`\"`	Comilla doble
`\r`	Retorno de carro	`\`*ooo*	Nº octal (hasta 3 dígitos)
`\t`	Tabulador horizontal	`\x`*hh...*	Nº hexadecimal
`\v`	Tabulador vertical		

Cuadro 2.1: Secuencias de escape

La secuencia de escape correspondiente al cierre de interrogación, ?, sólo es necesaria en cadenas de caracteres cuando hay dos cierres de interrogación seguidos, pues formarían un *trígrafo*, que es una forma alternativa de escribir ciertos caracteres necesarios en el lenguaje pero que quizá no estén disponibles en el teclado o conjunto de caracteres de determinada máquina. La secuencia de escape del apóstrofo o comilla simple sólo es necesaria en literales *char*: `'\''`, y análogamente la de la comilla doble sólo en literales de cadena de caracteres. Las dos últimas formas de la tabla 2.1 permiten escribir cualquier carácter del conjunto de caracteres de la máquina en su forma numérica, como un número octal o hexadecimal, bien sea el carácter imprimible o no. Se admiten sólo hasta 3 dígitos octales y un número indeterminado de dígitos hexadecimales; en este caso la equis al lado de la barra inclinada ha de ser minúscula, pero las letras de la A a la F para denotar los dígitos hexadecimales del 10 al 15 pueden ser mayúsculas o minúsculas.

Algunos compiladores admiten otras secuencias de escape, pero no son estándar. Por ejemplo, `\e` para el carácter Esc (*escape*), pero este carácter no existe en algunos juegos de caracteres, por eso no está incluido en las secuencias estándar. Una forma alternativa de ponerlo sería `\33` o `\x1b`.

Una diferencia con el C: en C no existen literales de tipo *char*, sino que por ejemplo `'A'` es de tipo entero (*int*). En C++ tal literal sí es de tipo *char*; esto se hizo necesario para la sobrecarga de funciones, que se verá más adelante.

Otro tipo carácter es el denotado por la palabra reservada `wchar_t`; el peculiar nombre viene de *wide character type*, o sea, tipo de caracteres anchos, y en C era un sinónimo de otro tipo entero básico, hecho con un `typedef` en la cabecera `<stddef.h>`. En C++ está incorporado en el lenguaje, pero para mantener la compatibilidad con el C, se ha dejado el mismo nombre. Representa el tipo para caracteres anchos, que no caben en un byte. Para nosotros no tiene ningún interés; donde sí lo puede tener es en países asiáticos con alfabetos enormes. Un literal de

tipo *wchar_t* se forma anteponiendo la letra L a un carácter o conjunto de ellos entre apóstrofos, como en `L'ab'`.

Enteros

Aunque dentro de los enteros podríamos meter también a los caracteres recién vistos, reservamos esta palabra para los otros tipos de números enteros denotados con la palabra reservada `int` y quizás algún modificador de signo o tamaño.

En cuanto al signo, se puede aplicar la palabra reservada `unsigned` para denotar que la cantidad se considera sin signo.

Esta palabra sirve sobre todo para tratar la memoria como un vector de bits y aplicar sobre el dato operadores de bits, o para representar el dato mediante campos de bits. Sin embargo, contra lo que se cree, utilizar un *unsigned* en vez de un simple *int* para ganar un bit más para representar números enteros positivos no suele ser una buena idea.

La palabra `signed` es redundante con los *int*s, pues todos tienen signo por omisión.

En cuanto al tamaño, la palabra reservada `short` indica un entero corto y `long` un entero largo.

El tipo *short* es una reminiscencia de los primeros tiempos del C y se emplea muy poco. No hay literales de tipo *short*, salvo utilizando un molde, como en `(short)16`.

El tipo *long* sí tiene más empleo, si bien en muchas máquinas el tamaño es el mismo que el del *int*.

Si se emplea un modificador de signo o tamaño, se puede omitir la palabra `int`; así por ejemplo, *signed* sería lo mismo que *int* o que *signed int* (por supuesto, en la práctica nadie emplea la palabra `signed` salvo con `char`); *long* es lo mismo que *long int*, o *unsigned short* es lo mismo que *unsigned short int*.

Los límites y otras propiedades de los tipos enteros, incluyendo los caracteres, se encuentran descritos en forma de macros del preprocesador en el fichero de cabecera `<climits>` o en forma de constantes o pequeñas funciones en `<limits>`.

Los literales pueden ponerse en tres bases: decimal, octal o hexadecimal. Los literales en decimal son los más comunes: una secuencia de dígitos, pero el primero no debe ser el 0 salvo que se esté representando el número cero. Para

poner un literal entero en base ocho, el primer dígito debe ser el 0, y para ponerlo en hexadecimal el primer dígito será el 0 y le seguirá la letra *equis* mayúscula o minúscula. Naturalmente, en un literal octal no se permiten los dígitos 8 ni 9, y en uno hexadecimal se pueden emplear las letras de la *a* a la *efe* mayúsculas o minúsculas.

Ejemplo:

decimal	0	2	63	83
octal	00	02	077	0123
hexadecimal	0x0	0X2	0x3F	0X53

> El empleo de las bases 8 y 16 para representar números se debería dejar para representar patrones de bits, donde son muy útiles (lamentablemente no se pueden poner literales en base 2). El empleo de estas notaciones para números *genuinos* (es decir, que no representen patrones de bits) puede llevar a sorpresas. Por ejemplo, `0xffff` puede ser -1 en una máquina donde el *int* se represente con complemento a 2 y tenga 16 bits, pero será 65535 si tiene 32 bits.
>
> Se puede forzar a que el tipo de un literal entero sea *long*, *unsigned* o ambos mediante los sufijos L o U respectivamente. Pueden ir combinados en cualquier orden, en mayúsculas o minúsculas (aunque no se recomienda la *ele* minúscula por su posible confusión con un 1). Por ejemplo: `1234L`, `0u`, `65535ul`.

Enumeraciones

Una enumeración es un tipo que puede contener un conjunto de valores especificado por el usuario. Una vez definida, se emplea en muchos aspectos como un tipo entero. Los miembros de una enumeración son constantes enteras con nombre; sus valores son los indicados mediante el signo igual seguido de un entero y si no se indican, toman el del miembro anterior incrementado en una unidad; si el primero no está inicializado, vale 0.

Un tipo enumerado puede no tener nombre; en ese caso la utilidad que tiene es la definición de sus miembros como constantes enteras, siendo ésta otra forma aparte de las macros del preprocesador y los `const`. Si tiene nombre, éste se convierte en el nombre del tipo; cada enumeración es un tipo distinto.

Recuerde que en C el nombre del tipo es la palabra `enum` más el nombre de la enumeración; en C++ basta con este último.

Ejemplo:

```
enum { tam = 100 };
// ...
if (a > tam) // ...

enum color { rojo, verde, azul }; // rojo == 0, verde == 1,
                                  // azul == 2
color c = rojo;
```

Las enumeraciones son útiles sobre todo para mejorar la legibilidad del código, para dar al usuario que lo lee una mejor idea de lo que se pretende.

Sin embargo no son lo mismo que en C. En C cualquier enumeración es exactamente igual que un tipo *int*; una variable de cualquier tipo enumerado se trata igual que si fuera *int*; no hay noción de rango y la conversión entre *int* y *enum* es trivial.

En cambio en C++ cada enumeración es un tipo distinto, la conversión desde entero a enumerado debe hacerse explícitamente, y cada enumeración tiene un rango: contiene todos los valores de los enumeradores redondeados a la siguiente potencia de dos, menos uno, lo que define el campo de bits más pequeño capaz de alojar todos los enumeradores. Si el entero que se va a convertir a un tipo enumerado no está en el rango, el resultado no está definido.

Ejemplo:

```
enum e1 { oscuro, claro };  // rango: 0..1
enum e2 { a = 3, b = 9 };   // rango: 0..15
enum e3 { min = -10, max = 1000000 }; // desde -1048576
                                      // hasta +1048575

enum ejemplo { x = 1, y, z = 4, e = 8 }; // rango: 0..15

ejemplo ej1 = 5;              // ERROR: 5 no es de tipo ejemplo
ejemplo ej2 = ejemplo(5);     // Bien: ejemplo(5) es de tipo
                              // ejemplo y está en el rango [0, 15]
ejemplo ej3 = ejemplo(z | e); // Bien, por lo mismo de antes:
                              // z | e vale 12
ejemplo ej4 = ejemplo(99);    // NO DEFINIDO: 99 está fuera
                              // del rango de 0 a 15
int i = ej3;                  // Bien: la conversión de enum a int
                              // sí está permitida: i vale 12.
```

El tamaño de un tipo enumerado, dado por el operador `sizeof`, es el de algún tipo entero que pueda alojar sus enumeradores, pero no es mayor que el tamaño de un *int* a menos que un enumerador no pueda representarse como un (*unsigned*) *int*. Por ejemplo, `sizeof(e1)` podría ser 1 ó 4, pero no 8 si `sizeof(int)` es 4.

Tipos de coma flotante

Representan números reales (en realidad, números en coma o punto flotante). El formato depende del sistema, aunque suele ser común el formato estándar IEEE-754–1985.

Como los enteros, estos tipos vienen en tres tamaños: `float` (abreviatura de *floating point*) para precisión simple, `double` para precisión doble y `long double` para precisión cuádruple o extendida. El significado exacto de cada uno depende del sistema; a la hora de tener que escoger, a menos que se sepa realmente lo que se está haciendo, lo mejor suele ser utilizar el tipo *double*.

> Las propiedades y límites de los números en coma flotante se describen en forma de macros del preprocesador en el fichero de cabecera `<cfloat>`, o en forma de constantes y pequeñas funciones en `<limits>`.

Los literales se forman con un conjunto de dígitos, un punto, otro conjunto de dígitos y opcionalmente una *e* mayúscula o minúscula y otro conjunto de dígitos quizá precedido por un signo + o −. Se puede omitir uno de los conjuntos de dígitos de alrededor del punto para indicar 0.

Los literales son de tipo *double*; para forzar las otras precisiones se emplean los sufijos *F* para *float* y *L* para *long double* respectivamente, en mayúsculas o minúsculas (como antes, evite la *ele* minúscula).

Ejemplo:

```
1.23   .23   0.23  1.  1.0  0.  .0  0.0   1.2e10  1.23E-15
3.14f  2.0F  2.997925e+08f  6.23E23L
```

Punteros

Los punteros en C++ son iguales que en C, con un par de salvedades.

Los punteros son variables que contendrán la dirección de un objeto (otra variable o una función). Se accede al objeto al que *apuntan*, es decir, cuya

dirección contienen, mediante el operador unario *desreferencia* o *indirección*, `*`; la dirección de un objeto se obtiene mediante el operador dirección, `&`. Para declarar un puntero se escribe el tipo base, un * y el identificador. El tipo base determina qué cantidad de memoria cogerá el compilador, y cómo tendrá que interpretarla, cuando se desreferencie el puntero.

Ejemplo:

```
int i = 0;
double d;

int* pe = &i;
double* pd;

*pd = 3.2;   // ERROR en ejecución, pd no apunta a nada
pd = &d;     // pd «apunta a» d
*pd = 3.2;   // Bien, ahora sí
pd = 0;      // NULL
cout << *pd; // ERROR, desreferencia del puntero nulo
```

Como en C, los punteros son muy «peligrosos», pues dan lugar a errores de ejecución difíciles de detectar, cuando se desreferencian punteros no inicializados, o que no tienen un valor asignado (esto es, no contienen una dirección válida, no apuntan a nada) o que contienen el valor 0 (nulo). Sin embargo son una herramienta indispensable.

Punteros genéricos Cuando el tipo base de un puntero es *void*, hablamos de punteros genéricos. Sirven como parámetros de funciones, para que puedan recibir o devolver en principio punteros de cualquier tipo; aunque en C++ no tienen mucho uso, pues el polimorfismo o la generalidad (*plantillas*) pueden hacer ese trabajo, son precisos para utilizar las bibliotecas de funciones de C.

Ejemplo:

La función de la biblioteca estándar de C (y esta biblioteca está incluida en la de C++)

```
void *memset(void *s, int c, size_t n);
```

rellena con el valor contenido en c (convertido a *unsigned char*) los primeros n bytes a partir de la posición de memoria contenida en s, y devuelve la

dirección de esa posición. Como no se sabe lo que hay almacenado en ella, se trabaja con punteros genéricos.

```
char v[30], *p;

p = (char*)memset(v, 0, sizeof v);
```

La función recibe como primer parámetro *v*; es decir, `&v[0]`, la dirección del primer elemento del vector, que se copia en el puntero genérico que hemos llamado *s* dentro de la función; como se ve, un puntero genérico admite cualquier otro tipo de puntero.

Sin embargo, como C++ es un lenguaje fuertemente tipado, a diferencia del C, no es posible la conversión implícita en el otro sentido y hay que modelar el tipo de retorno al puntero concreto, en este caso *char**.

Un puntero genérico no tiene información del tipo base. Al ser C un lenguaje muy flexible y poco tipado, que *confía* bastante en el programador (y aún confiaba más antes de su normalización), no hace falta modelar el tipo; ya sabrá el programador lo que está haciendo. Como esto es una fuente potencial de errores, y el C++ es un lenguaje fuertemente tipado, el programador tiene que ser explícito sobre la conversión desde puntero genérico a puntero concreto: se le obliga a que escriba lo que quiere hacer.

El puntero nulo En C, para expresar que un puntero no tiene que apuntar a ningún sitio válido, se utiliza el valor 0; esto suele indicar el fin de una lista u otra estructura dinámica de datos, o un valor anómalo de tipo puntero devuelto por una función, para indicar algún tipo de fallo. En vez de un simple 0 es muy común emplear una macro estándar definida en muchas cabeceras: *NULL* . Esta macro suele estar definida así:

```
#define NULL (void*)0
```

Es decir, el cero modelado a puntero genérico. Pero ya hemos visto que en C++ no podemos pasar de puntero genérico a otro tipo puntero sin un molde, por lo que esto sólo causaría problemas:

```
int* p = NULL;       // ERROR, NULL es void*
int* q = (int*)NULL; // Bien, pero tedioso.
```

Por ello en las cabeceras de C, normalmente diseñadas tanto para C como para C++, se suele definir *NULL* así:

```
#ifdef __cplusplus
#define NULL 0
#else
#define NULL (void*)0
#endif
```

Y es que en C++ el compilador garantiza que el simple 0 es un valor válido de cualquier tipo puntero, y se puede emplear siempre que se espere una expresión de cualquier tipo puntero; así que *NULL* es superfluo y en C++ no hay costumbre de usarlo; el ejemplo anterior se pone mejor así:

```
int* p = NULL; // Peligroso: bien sólo si NULL == 0
int* q = 0;    // Bien, lo mejor
```

> En realidad, en C también el *NULL* es superfluo y se podría emplear 0 en su lugar, pero es una costumbre arraigada. Sólo no se debe utilizar 0 al pasarlo como parámetro puntero a una función no declarada, o declarada sin el prototipo, o con parámetros no especificados, pues el compilador no tiene información en ese caso de si el parámetro es un entero o un puntero. Pero tampoco hace falta la macro *NULL* : lo mejor sería el 0 modelado al tipo puntero concreto.

Referencias

Ya hemos hablado de ellas en §2.2.5; son como punteros constantes que se desreferencian automáticamente. Se declaran como los punteros, pero sustituyendo el `*` por el `&`; su uso prácticamente se restringe a las funciones, como parámetros y valores de retorno. Deben inicializarse siempre, y esto es precisamente lo que ocurre cuando llamamos a una función.

Las referencias son imprescindibles para poder definir el constructor de copia y casi indispensables para poder redefinir el operador de asignación; estos conceptos, o sea, la definición de la copia de objetos definidos por el usuario, se verán en detalle en el capítulo correspondiente, más adelante.

También conviene muchas veces que las funciones devuelvan objetos por referencia, para evitar la copia que se produciría si se devolvieran por valor. ¿Cómo impedir entonces que se pueda asignar a la función, y por tanto se modifique ese objeto devuelto por ella? Simplemente definiendo el tipo de devolución de la función como una referencia `const`.

Vectores de bajo nivel

Seguimos con los tipos incorporados en el lenguaje; los vectores de bajo nivel son tipos agregados, formados por varios elementos del mismo tipo almacenados en memoria uno a continuación de otro, sin huecos. Decimos «de bajo nivel» para distinguirlos del tipo *vector* definido en la biblioteca estándar de C++, que se estudiará en §2.3.4; éstos serían vectores «de alto nivel».

Se define un vector añadiendo tras el identificador entre corchetes una constante entera positiva, que determina el tamaño. Se accede a cada elemento mediante el identificador seguido de una expresión entera entre corchetes, que debe evaluarse a un valor dentro del rango, empezando siempre por 0 para el primer elemento y acabando en tamaño menos una unidad para el último. Es posible indexar por el tamaño, es decir, un elemento más allá del último, siempre que no se modifique, puesto que no forma parte del vector; esto es útil a veces en bucles.

Ejemplo:

```
const int t = 10;
int j = 20;

int v[5];
static long double x[t];
int* pv[j];        // ERROR, j no es constante
float vf[j - t];   // ERROR, la expresión no es constante
char vc[t + 1];    // Bien, expresión constante positiva: 11
char vv[t - 10];   // ERROR, tamaño 0
unsigned char b[]; // ERROR, tamaño desconocido
```

Para inicializar un vector se escriben entre llaves y separados por comas los inicializadores de los elementos, en orden; si se omiten algunos, se inicializan a 0. Si se inicializa un vector con todos sus elementos especificados, entonces es posible omitir el tamaño: el compilador lo calcula automáticamente. También es posible omitir el tamaño cuando se declara, pero no define, un vector, pues entonces estamos diciendo que el vector está definido en otro sitio, donde sí tendrá el tamaño.

Ejemplo:

```
int v1[5] = { -2, -1, 0, +1, +2 };
int v2[] = { -2, -1, 0, 1, 2 };    // Bien, tamaño: 5
```

```
int v3[];                          // ERROR, tamaño desconocido
extern int v2[];                   // Bien, sólo declaración
signed char v4[30] = { 0 };        // todos los 30 elementos a 0
double v5[5] = { 2.1, 3.2 };       // los 3 últimos a 0.0
```

Los vectores no se pueden pasar directamente a funciones, ni como referencias; aunque por conveniencia o claridad se pueden escribir los parámetros formales de las funciones, en prototipos o definiciones, como vectores con corchetes. El identificador de un vector, empleado en una expresión, equivale siempre a la dirección de su primer elemento, por lo que el tipo decae de «vector de N elementos de tipo T» a «puntero a T».

Ejemplo:

```
int a[14];

void fu1(int v[5]);
void fu2(int v[]);
void fu3(int *v);

fu1(a);
fu2(a);
fu3(a);
```

En todos los casos las funciones reciben lo mismo: la dirección de comienzo de a, o sea, `a == &a[0]`. Esta dirección se copia en el *puntero* v, **que lo es en los tres casos**. En *fu1()*, el número 5 no se tiene en cuenta, se podría haber puesto cualquier otro; lo normal es no poner ninguno, como en *fu2()*.

Existe una estrecha relación entre punteros y vectores. Cuando un puntero apunta a un vector, la suma o resta de un entero se hace por un factor de multiplicación igual al tamaño de cada elemento, así como la resta de dos punteros que apunten a un mismo vector es el número de elementos entre ellos; de forma que

```
int v[5], i, *p;

p = v;   // p = &v[0], p apunta a v[0], *p es v[0]
p + 1;   // equivale a &v[1]
v + 1;   // equivale a &v[1]
p[1];    // equivale a v[1]
p++;     // p apuntará a v[1], *p será v[1]
--p;     // p apunta ahora de nuevo a v[0]
```

```
v++;     // ERROR, v no es modificable, es la dirección del vector
p += 4;  // p apunta al último elemento del vector, *p == v[4]
p - v;   // da 4, el número de elementos entre ellos
```

> El tipo resultante de la resta de dos punteros es un entero con signo dependiente del sistema; para que en todos tenga el mismo nombre se define el tipo concreto con el nombre *ptrdiff_t* en la cabecera `<cstddef>`.

El operador índice, o sea, los corchetes, no hace más que una suma desde la dirección base (un desplazamiento) y una desreferencia, cumpliéndose para un vector v que

```
v == &v[0]     v[i] == *(v + i)    &v[i] == v + i
```

por lo que es lo mismo trabajar con notación de punteros que con índices.

Matrices Las matrices o tablas de más de una dimensión no existen, sino que se forman a partir de vectores de vectores: vectores cuyos elementos son a su vez otros vectores. Se definen poniendo los tamaños uno a continuación de otro, cada uno encerrado entre corchetes:

```
int a[3][2];       // 3 filas x 2 columnas
int b[3, 2];       // ERROR, operador «coma» con constantes
int c[2][3] = {    // vector de 2 elementos, cada uno de
     { 1, 2, 3 },  // ellos es un vector de 3 elementos int;
     { 4, 5, 6 },  // o sea: matriz de 2x3,
   };              // 2 filas y 3 columnas
int d[2][3] = { 1, 2, 3, 4, 5, 6 }; // igual, pero menos explícito
int e[2][3][4];
```

Estos vectores, como su nombre indica, son *de bajo nivel*: indispensables en C, en C++ la norma general es **no utilizarlos**, y ampliando esta premisa, utilícense sólo en optimizaciones de código, cuando la eficiencia sea muy importante; o en implementaciones de clases, a bajo nivel. ¿Por qué todo esto? Porque los vectores de bajo nivel son inherentemente peligrosos: podemos pasarnos del rango tranquilamente sin que el compilador nos diga nada, y eso produce errores muy sutiles y difíciles de detectar, lo mismo que ocurre con los punteros. Muchos fallos de seguridad en programas incluso comerciales son debidos al uso (incorrecto, desde luego) de estos vectores y a funciones que trabajan con ellos y no controlan el posible sobrepaso de sus límites.

En C++, utilice mejor la clase *vector* de la biblioteca estándar, mucho más cómoda de emplear.

Cadenas de caracteres de bajo nivel

Las cadenas de caracteres de bajo nivel, para distinguirlas de las que proporciona la biblioteca estándar, son un caso especial de vectores (§2.3.2) donde el tipo base es el carácter (*char*); como los vectores de bajo nivel, tienen una longitud máxima, pero además se considera que la cadena de caracteres acaba en el carácter cuyo código numérico es 0; o sea, el literal `'\0'` o `'\0x0'`. Esto es un convenio para las funciones de la biblioteca estándar de C que tratan con estas cadenas; dichas funciones esperan encontrar este carácter terminador y no funcionarán bien si no lo encuentran.

El único soporte que da el lenguaje en sí para las cadenas de caracteres es en los literales de este tipo, que se forman encerrando una ristra de caracteres entre comillas dobles; el compilador automáticamente añade el `'\0'` final.

Ejemplo:

```
char v1[] = { 'a', 'b', '\027' };    // ¡NO es una cadena!
char v2[] = { 'C', '+', '+', '\0' };
char v3[] = "C++";
char v4[] = "";
char* p1 = "Bjarne Stroustrup";
```

v1 es un vector de bajo nivel de 3 elementos de tipo *char*; *no* contiene una cadena de caracteres, puesto que no acaba en el carácter `'\0'`.

v2 es un vector que sí contiene una cadena de caracteres, puesto que el último elemento es el terminador 0.

v3 es como *v2*, pero se inicializa con un literal «cadena de caracteres», lo que es más cómodo que el caso anterior. El compilador añade automáticamente el carácter `'\0'` al final.

v4 es un vector inicializado con la cadena de caracteres vacía o nula. Tiene de tamaño uno, el elemento `'\0'` solamente, aunque la *longitud* de la cadena es 0, pues el terminador no se cuenta.

p1 es un puntero que apunta a una cadena de caracteres con el nombre del inventor del lenguaje C++.

Como se ve, las cadenas de caracteres pueden almacenarse como un vector de *char*, o un puntero puede apuntar a uno de dichos vectores o a un literal, como en el caso anterior. En este caso, no está definido que se pueda modificar el literal; no se debe, pues el literal es una constante, aunque muchos

compiladores lo permiten[4]. El tipo de un literal como `"Hola"` es «vector de 5 *char*», pero al emplearse en una expresión, por ejemplo al pasarse a una función, este tipo se convierte, *decae*, en «puntero a 5 caracteres constantes»; o sea, *const char**.

> Un puntero, tanto en C como en C++, no contiene información de si apunta a un solo objeto o a un vector de ellos, ni a cuántos; de ahí que un *char** puede leerse como «puntero a *char* o a carácter», «puntero a caracteres» o «cadena de caracteres», según el contexto.

Como los vectores, no se pueden asignar, comparar o copiar cadenas de caracteres; hay que hacerlo directamente elemento a elemento o mediante funciones de la biblioteca estándar, declaradas en `<cstring>`. Y como se dijo con los vectores, la norma general es *no emplearlos* salvo cuando la eficiencia sea muy importante, o en implementaciones de clases, a bajo nivel. En su lugar, deben emplearse objetos de la clase *string*, definida en la biblioteca estándar de C++, que tienen muchas propiedades convenientes, como se verá en §2.3.3.

Estructuras

Las estructuras nos permiten almacenar varias variables como una sola. Las variables que componen una estructura se llaman *miembros* o *campos* y pueden ser de cualquier tipo salvo el de la propia estructura. Un tipo estructura se define (normalmente en un fichero de cabecera para poder compartir su definición entre varias unidades de traducción o ficheros) mediante la palabra reservada `struct` seguida de un identificador opcional; en C, este identificador define al nuevo tipo pero tiene que ir precedido de la palabra `struct`; en C++, el identificador actúa como una nueva palabra reservada de tipo y no hace falta `struct`. Si no se pone el identificador, la estructura es anónima y no puede referenciarse más tarde, por lo que tras su definición tendrá que haber una lista de declaraciones de variables de su tipo. La definición de la estructura propiamente dicha consiste en una serie de declaraciones de variables (los miembros o campos) entre llaves.

Como las demás variables, una de tipo estructura se puede inicializar mediante un inicializador de estructuras, que consiste en una serie de valores para cada campo, por orden, separados por comas y encerrados entre llaves.

[4] Pero aunque el compilador lo permita, no se debe hacer: una de las optimizaciones frecuentes que realiza el compilador es la llamada «fusión de literales de cadena», que consiste en almacenar los literales cadena iguales en el mismo sitio; por tanto, la modificación de un literal podría hacer que los que fueran iguales también se modificaran.

Ejemplo:

```
struct { int i[5]; } v; // anónima

struct Entrada {
  char *nombre;
  unsigned long numero;
}; // observe el ';': esto no es una sentencia compuesta

struct Nodo {
  void *dato;
  Nodo *sgte;
};

struct Dir {
  char *nombre;
  unsigned numero_i;
} d1, d2;

Entrada lista[100], f = { "Fulano", 888888 }, *p = &f;
lista[0] = f;
Nodo cab = { 0, 0 };
```

En realidad se puede hablar mucho más de las estructuras, pues son la base del C++; las *clases* son prácticamente iguales que ellas, al menos bajo un punto de vista sintáctico (de hecho una de las dos palabras, `struct` o `class`, es superflua); pero en la práctica es de buen estilo reservar la palabra `struct` para las estructuras *como en C*; es decir, para almacenar variables relacionadas que forman una estructura de datos sin más.

Dos estructuras del mismo tipo se pueden copiar (esto implica pasar a funciones por valor, o devolver éstas) o asignar, se puede tomar la dirección de una estructura; pero lo que es fundamental en ellas es acceder a los campos. Para ello disponemos del operador de selección de miembro `.` (*punto*) y del de selección de miembro a través de puntero, `->` (llamado *flecha*, pero en realidad es un signo *menos* y un *mayor que* juntos).

Ejemplo:

```
f.numero = 4444u;
p->numero++;      // equivale a (*p).numero++;
```

Punteros a miembros Un puntero es una variable que contiene la dirección de un dato o una función, de forma que uno puede cambiar en tiempo de ejecución lo que un puntero seleccione. Un *puntero a miembro* sigue el mismo esquema, salvo que lo que se selecciona es una dirección dentro de una clase. El problema es que no existe una tal «dirección» dentro de una clase: seleccionar un miembro de una clase significa desplazarse dentro de ella. Así pues, para producir una verdadera dirección hay que combinar ese desplazamiento con la dirección de comienzo de un objeto de la clase en cuestión. La sintaxis de los punteros a miembros refleja este hecho, requiriéndose la selección de un objeto (`.` o `->`) al mismo tiempo que se desreferencia el puntero a miembro (`*`).

Para entender esto, veamos un ejemplo muy simple. Consideremos una estructura sencilla, un objeto de ese tipo y un puntero:

```
struct Simple { int a; void f(int); } os, *ps = &os;
```

Ya sabemos que podemos seleccionar un miembro mediante los operadores *punto* y *flecha*, así:

```
os.a;
ps->a;
```

Supongamos ahora que tenemos un entero i y un puntero a entero pi que apunte a él:

```
int i, *pi = &i;
```

Para acceder al objeto apuntado por pi, desreferenciamos el puntero con el operador de indirección:

```
*pi = 4;
```

Finalmente, consideremos lo que ocurre si tenemos un puntero que apunte a algo de dentro de un objeto de clase. Para acceder a lo que esté apuntando, tenemos que desreferenciarlo con `*`, como antes. Pero realmente es un desplazamiento dentro del objeto de clase, por lo que también tenemos que referenciar dicho objeto. De ahí los nuevos operadores `.*` y `->*`. Siguiendo el ejemplo de la clase o estructura anterior y siendo pm un puntero al miembro a de dicha estructura:

```
os.*pm = 47;
ps->*pm = 47;
```

Observe que tanto `.*` como `->*` son dos nuevos operadores de C++; no son dos pares de operadores combinados. Ambos forman lexemas que no pueden escribirse por separado. Es un error de sintaxis algo como:

```
os . * pm = 47;   // ERROR
ps -> *pm = 47;   // ERROR
```

La siguiente cuestión es cómo definir *pm*. Como cualquier puntero, tenemos que especificar el tipo base y emplear * en la declaración. Pero también tenemos que decir con qué clase de objetos se empleará, por lo que hay que anteponer al * el nombre de la clase y el operador de ámbito:

```
int Simple::*pm;
```

Por supuesto, *pm* puede inicializarse, o puede asignársele un valor más tarde. ¿Cómo se especifica el inicializador? De una curiosa forma: «la dirección del miembro de la clase»; quedaría así:

```
int Simple::* pm = &Simple::a;
```

Decimos que es curioso porque no se inicializa con ningún objeto en particular; de hecho no existe tal cosa como «la dirección» de `Simple::a`, puesto que no es un objeto.

Normalmente, los miembros de datos forman parte de la parte privada de una clase y por tanto no son accesibles. Es más normal acceder a métodos de una clase mediante los punteros a miembros. Así como

```
void (*pf)(int);
```

declara *pf* como un puntero a función que recibe un entero y no devuelve nada, ahora definiríamos un puntero a una función miembro *Simple* con un prototipo como el de *f()* así:

```
void (Simple::*pf)(int);
```

Y si quisiéramos inicializarlo para que apuntara al método *f()*, la declaración completa sería:

```
void (Simple::*pf)(int) = &Simple::f;
```

A diferencia de con los punteros a funciones, el `&` es aquí obligatorio:

```
pf = Simple::f;  // ERROR de sintaxis
```

El valor de un puntero es poder cambiar el sitio al que apunta en tiempo de ejecución. Lo mismo ocurre con un puntero a miembro. Como hemos dicho antes, lo normal es tener punteros a miembros funciones, puesto que los atributos suelen ser privados. Para evitar en lo posible la complicada sintaxis de los punteros a miembros se suelen utilizar cláusulas `typedef`.

Ejemplo:

pmem.cpp

```
1  // Ejemplo de punteros a miembros
2  class Cosa {
3  public:
4    void f(int);
5    void g(int);
6    void h(int);
7    void i(int);
8  };
9
10 void Cosa::h(int) {}
11
12 typedef void (Cosa::*Pmiembro)(int);
13
14 int main()
15 {
16   Cosa c;
17   Cosa* pc = &c;
18   Pmiembro pm = &Cosa::h;
19   (c.*pm)(1);
20   (pc->*pm)(2);
21 }
```

Así el usuario de la clase puede seleccionar en tiempo de ejecución a qué método llamar sin utilizar su nombre directamente. Pero para liberarlo de esta complicada sintaxis se podrían incluir los punteros a miembros *dentro* de la clase, de forma que el usuario simplemente elegiría el método mediante un número:

pmem2.cpp

```
1  // Punteros a miembros dentro de una clase
2  #include <iostream>
3  using namespace std;
```

```
4
5  class Cosa {
6    void f(int a) const { cout << "Cosa::f(" << a << ")\n"; }
7    void g(int a) const { cout << "Cosa::g(" << a << ")\n"; }
8    void h(int a) const { cout << "Cosa::h(" << a << ")\n"; }
9    void i(int a) const { cout << "Cosa::i(" << a << ")\n"; }
10   static const int num = 4;        // /* Equivalente: */ enum { num = 4 };
11   void (Cosa::*ptrfun[num])(int) const;
12 public:
13   Cosa()
14   {
15     ptrfun[0] = &Cosa::f;
16     ptrfun[1] = &Cosa::g;
17     ptrfun[2] = &Cosa::h;
18     ptrfun[3] = &Cosa::i;
19   }
20   void seleccionar(int i, int j)
21   {
22     if (0 > i || i >= num) return;
23     (this->*ptrfun[i])(j);
24   }
25   int numero() const { return num; }
26 };
27
28 int main()
29 {
30   Cosa c;
31   for (int i = 0; i < c.numero(); ++i)
32     c.seleccionar(i, 47);
33 }
```

La inicialización de los punteros a miembros en el constructor parece recargada; ¿no se podría haber puesto sencillamente

```
ptrfun[1] = &g;
```

ya que *g* es un miembro de la propia clase y por tanto está en el mismo ámbito? No, porque hay que seguir la sintaxis de los punteros a miembros, que exigen que el inicializador sea de la forma *clase*::*miembro*.

Del mismo modo, podría parecer a primera vista que sobra, como otras veces, `this->` de

```
(this->*ptrfun[i])(j);
```

pero no es así; en primer lugar, `->*` es un solo operador y no puede omitirse

`->`; en segundo lugar, ese operador exige un operando en la parte izquierda que debe ser un puntero a un objeto de clase.

Campos de bits Los campos de bits son campos de estructuras de tipo (casi siempre *unsigned*) *int*, donde se especifica el tamaño del campo en bits; para ello, tras el identificador del campo, se pone un signo de dos puntos y un literal entero que especifica el tamaño antedicho. Se puede omitir el identificador, en cuyo caso no podremos acceder a él pero ocupa su sitio; y un tamaño 0 indica alinear al siguiente límite de palabra.

Ni que decir tiene que los campos de bits se utilizan en programación a muy bajo nivel. No sólo eso, sino que además son normalmente preferibles los operadores de bits, pues los campos de bits hacen depender al programa fuertemente de la máquina donde se ejecute.

Ejemplo:

```
struct Bits {
  unsigned msb: 1;
  unsigned: 6;
  unsigned lsb: 1;
};

Bits byte;
// ...
byte.msb = 1;
```

El primer campo tiene 1 bit de longitud y se llama *msb*. El segundo tiene 6 bits de longitud y no tiene nombre (sirve para dejar 6 bits de espacio entre el *msb* y el *lsb*). El tercero se llama *lsb* con un bit de longitud.

Uniones

Sintácticamente, las uniones son casi como las estructuras, salvo que la palabra reservada `struct` se sustituye por `union`, pero sin embargo hay una diferencia fundamental: mientras los campos de una estructura se almacenan uno a continuación del otro, quizás con algún que otro hueco entre ellos debido a requisitos de alineación en memoria, todos los campos de una unión se almacenan a partir de la misma dirección de memoria. Esto implica que en cada momento sólo tiene sentido almacenar un solo campo, y el programador debe encargarse de saber cuál y actuar en consecuencia. Una unión sólo

puede inicializarse por tanto a través de un solo campo, que es el primero que se escriba.

Por supuesto el empleo de las uniones en C++ está restringido a la programación a bajo nivel; aunque a veces se han utilizado para ahorrar memoria, guardando varias variables (de las que se sabe que en cada momento no hará falta más de una) en un solo sitio.

Por lo demás, todo es igual: para acceder a un miembro se utiliza uno de los operadores punto o flecha; se pueden asignar, copiar, etc. Una diferencia con C es que en C++ puede haber uniones anónimas y, sin emplear variables de ese tipo unión, se pueden utilizar directamente sus campos como variables; eso sí, variables almacenadas en el mismo sitio. Si una unión como ésta es global, debe ser `static`, para que tenga enlace interno, o alcance de fichero.

> En C++, una `union` puede tener funciones miembro, constructores y destructores, como una `class` o `struct`, pero esto se utiliza poquísimo. Las funciones miembros se verán en el capítulo sobre las clases.

Muchas veces las uniones se emplean como miembros de estructuras, en las cuales un miembro mantiene información de qué campo de la unión está *activo* en cada momento.

2.3.3. Cadenas de caracteres: *string*

Como se ha dicho, las cadenas de caracteres de bajo nivel, si bien tremendamente eficientes, son algo tediosas de utilizar y propensas a cometer errores. Normalmente, lo mejor es utilizar el tipo llamado *string*, definido en la cabecera estándar `<string>`, y en el espacio de nombres *std*, como todos los elementos de la biblioteca estándar. No hay que confundir la cabecera `<string>`, donde se define el tipo *string*, con `<cstring>` o `<string.h>`, donde se definen las funciones que trabajan con las cadenas de caracteres de bajo nivel, las de C.

Inicialización Las cadenas de caracteres de las que se está hablando pueden construirse (*inicializarse*) de varias maneras: mediante una cadena de bajo nivel o parte de ella, mediante otra cadena o parte de ella, mediante una secuencia de caracteres; pero no por un carácter o un entero, porque no tiene sentido una cadena vacía de longitud $x > 0$; es decir, si se pudiera definir `string s(5);` se supone que se quiere construir un *string* de longitud 5 caracteres, pero sin almacenar ninguno. Esto no tiene sentido con los *string*s, porque su longitud es siempre el número de caracteres que almacena en un momento dado.

EJEMPLO:

```
string s1("Hola");     // con una cadena de bajo nivel
string s2 = "a todos"; // lo mismo, de otra forma
string vacia;          // la cadena vacía
string nada = "";      // otra vacía
string copia = s1;     // Con la copia de otro string
string a(10u, 'A');    // AAAAAAAAAA
string mar(s1, 1, 3);  // ola

string err1(7);        // ERROR
string err2 = 7;       // ERROR, no hay conversión definida
string err3 = 'a';     // ERROR, por lo mismo
string s3 = "a";       // esto sí: "a" es una cadena de C
```

Longitud La longitud de una cadena, esto es el número de caracteres, se obtiene mediante las funciones miembro *string::length()* o *string::size()*; en C++, como se verá, los tipos definidos por el usuario (como es *string*) pueden incluir funciones como miembros, a las que se accede como a otros miembros de una estructura, con los operadores de selección de miembro `.` o `->`.

EJEMPLO:

Siguiendo con las definiciones del ejemplo anterior, la sentencia

```
cout << s1.length() << ' ' << s2.size() << endl;
```

debería mostrar en la salida estándar los números 4 y 7.

Estas cadenas de caracteres no tienen por qué almacenar ningún carácter terminador, como las cadenas de bajo nivel almacenaban el `'\0'`. La longitud es en cada momento el número de elementos que aloje.

Acceso Para acceder a un carácter determinado de un *string* podemos emplear el operador índice `[]` o la función miembro *string::at()*. La diferencia es que con `[]` no se comprueban los límites; con *string::at()* sí. De modo que los corchetes son más adecuados si ya estamos seguros de que no nos vamos a pasar del límite; además son obviamente algo más eficientes que la operación *string::at()*. Si empleamos esta última y nos salimos de rango, se lanza una excepción *out_of_range*; las excepciones se estudiarán en el capítulo 5.

Ejemplo:

```
string refran = "Cuando el diablo su rabo vende, "
                "él se entiende.";

char inicial = refran[0];         // el primer carácter: 'C'
char ultimo = refran[refran.length() - 1]; // el último: '.'

for (int i = 0; i < refran.length(); i++)
    refran[i] = toupper(refran[i]);

refran[100] = 'a'; // MAL: accede a una zona de memoria no
    // reservada; seguramente error en tiempo de ejecución
refran.at(100) = 'a';       // también mal, pero lanza una
                // excepción: comportamiento bien definido
```

Dentro del `for`, si lo hemos escrito bien, estamos seguros de que no nos podemos pasar del rango, así que podemos emplear con seguridad el operador `[]`, que es más eficiente. La función de C *toupper()*, declarada en `<cctype>`, devuelve en mayúscula el carácter que recibe.

Asignación y copia Estas cadenas pueden asignarse unas a otras. Cuando una cadena se asigna a otra, se tienen dos copias de la asignada. Se permite la asignación de un carácter, aunque no la inicialización.

Ejemplo:

```
// string s = 'a';  // ERROR
string s;
s = 'a';            // Bien

string cpp = "C++";
s = cpp;            // 2 copias de "C++"
s = s;              // auto-asignación: inútil pero inocua

string f(string);
```

La función *f()* recibe un *string* por valor y devuelve otro de la misma manera. Aunque válido, esto no suele ser conveniente; utilice en estos casos referencias, constantes si la función no ha de modificar su parámetro.

Concatenación Una de las principales operaciones con cadenas es añadirle otra al final. A un *string* le podemos añadir otro, una cadena de bajo nivel o un simple carácter. Para esto se redefinen los operadores + y +=.

Ejemplo:

```
string s1 = "Hola", s2 = "a todos";
string saludo = s1 + ' ';
saludo += s2;     // saludo: Hola a todos

string nombre_completo(const string& nombre,
                       const string& apellido1,
                       const string& apellido2)
{
  return "Nombre completo: " + nombre + ' ' +
         apellido1 + ' ' + apellido2;
}
```

Entrada y salida Entre las principales operaciones con *string*s están éstas. Los operadores << y >> están redefinidos o sobrecargados para trabajar con *string*s.

Ejemplo:

hola.cpp

```
1  // Un saludo más personal
2
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9    string tu;
10
11   cout << "¿Cómo te llamas? ";
12   cin >> tu;
13   cout << "Hola, " << tu << ", bienvenido al C++." << endl;
14 }
```

Como se ve, los *string*s tienen la interesante propiedad de expandirse lo que haga falta. Al principio, *tu* es una cadena vacía, pero se expande sin

problemas para alojar los caracteres que introduzcamos desde la entrada estándar. Esto no ocurre con las cadenas de bajo nivel, que son de una longitud fija y dan muchos problemas de desbordamiento si no andamos con mucho cuidado:

holac.cpp

```
1  // Un saludo más personal sin utilizar string's
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8    char tu[20];
9
10   cout << "¿Cómo te llamas? ";
11   cin >> tu;  // problemas si metemos más de 20 caracteres.
12   cout << "Hola, " << tu << ", bienvenido al C++." << endl;
13 }
```

Como con las cadenas de bajo nivel, el operador >> lee un conjunto de caracteres delimitado por blancos; o sea, se salta los espacios en blanco iniciales que pudiera haber, y se para al encontrar un blanco. En los ejemplos anteriores, la entrada

```
Carlos Jesús
```

produciría la salida:

```
Hola, Carlos, bienvenido al C++.
```

Si queremos leer una línea entera podemos emplear mejor la función *getline()*:

Ejemplo:

hola2.cpp

```
1  // Un saludo más personal, versión 2, con getline()
2
3  #include <iostream>
4  #include <string>
5  using namespace std;
```

```
6
7  int main()
8  {
9    string tu;
10
11   cout << "¿Cómo te llamas? ";
12   getline(cin, tu);
13   cout << "Hola, " << tu << ", bienvenido al C++." << endl;
14 }
```

Esta función admite otro parámetro, que es el delimitador; si no se proporciona, toma el fin de línea: `'\n'`; este delimitador se quita del flujo y no se guarda en la cadena, lo que es lo más cómodo casi siempre.

Subcadenas La operación *string::substr()* devuelve una copia de la subcadena indicada por sus parámetros: un índice o posición en el *string*, empezando por 0, y la longitud de la cadena deseada.

La operación *string::replace()* reemplaza una subcadena (definida por los dos primeros parámetros que son la posición y la longitud) por un valor dado como tercer parámetro, que es otra cadena de caracteres que no tiene por qué ser del mismo tamaño que la subcadena a reemplazar.

Ejemplo:

```
string nombre_compl("Cristobalito Gazmoño");
string nombre_pila = nombre_compl.substr(0, 12); // Cristobalito
string apellido = nombre_compl.substr(13, 7);    // Gazmoño
nombre_pila.replace(5, 7, "óbal");               // Cristóbal
```

Conversión a cadenas de bajo nivel Podemos convertir una cadena de caracteres de bajo nivel o al estilo C, es decir, un vector de caracteres acabado en `'\0'`, a *string*, como hemos visto; pero no existe una conversión implícita de *string* a cadena de bajo nivel; para ello debemos emplear una función miembro llamada *string::c_str()*. Esta función extrae los caracteres guardados en un *string*, les añade el terminador `'\0'` y los guarda en una zona de memoria perteneciente al *string* (por lo que tendremos que hacer una copia si queremos modificarlos).

> Existe también la función *string::data()* que es igual pero que no añade el terminador, por lo cual nos proporciona un simple vector de caracteres, siendo mucho menos útil que *c_str()*.

Esta función es útil si debemos emplear una función que reciba una cadena de bajo nivel; por ejemplo, alguna función de la biblioteca de C:

Ejemplo:

```
#include <cstdlib>
#include <string>
using namespace std;
// ...
string n = "12345";
int i = atoi(n.c_str());
```

Normalmente, si usamos *string::c_ str()* con mucha frecuencia es que estamos haciendo algo mal; deberíamos buscar una alternativa y trabajar con *string*s, o sobrecargar la función para que trabaje con *string*s.

Comparación Dos cadenas pueden compararse con los operadores habituales. Se comparan los contenidos, por supuesto, no como en el caso de las cadenas de C, donde se comparan las direcciones de memoria y hay que emplear funciones de biblioteca. Se pueden comparar dos *string*s entre sí, o un *string* con un literal.

Ejemplo:

```
void magia(const string& mago)
{
  if (mago == "Merlín")
  // ...
}
```

Si *mago* hubiera sido un *const char**, es decir, una cadena de bajo nivel, la comparación anterior resultaría en la de la dirección de comienzo de la cadena *mago* (o sea, `&mago[0]`) con la del literal `"Merlín"`. Como evidentemente siempre serán distintas, la comparación daría `false`. Tendría que haberse puesto en tal caso

```
if (strcmp(mago, "Merlín") == 0) // son iguales...
```

Búsqueda Existe una gran variedad de funciones miembro *string::find()* para buscar caracteres y subcadenas dentro de un *string*. Se resumen en el siguiente ejemplo, donde *string::size_type* es un tipo entero sin signo definido dentro de *string* (de ahí la cualificación `string::`) dependiente de la implementación. Cuando alguna de estas funciones fallan en la búsqueda, devuelven la constante *string::npos*, que es una posición ilegal, por lo que no debe utilizarse como tal en una operación con el *string*.

Ejemplo:

Como ejercicio, intente averiguar el significado de cada función a partir de su resultado, antes de leer la explicación.

```
string s = "accdcde";
// índices: 0123456
string::size_type i[7];

i[0] = s.find("cd");              // i[0] == 2
i[1] = s.rfind("cd");             // i[1] == 4
i[2] = s.find_first_of("cd");     // i[2] == 1
i[3] = s.find_last_of("cd");      // i[3] == 5
i[4] = s.find_first_not_of("cd"); // i[4] == 0
i[5] = s.find_last_not_of("cd");  // i[5] == 6
i[6] = s.find('h');               // i[6] == string::npos
```

- *i[0]* vale 2 porque *s[2]* contiene `'c'` y *s[3]* contiene `'d'`.
- *i[1]* vale 4 porque *s[4]* contiene `'c'` y *s[5]* contiene `'d'`.
- *i[2]* vale 1 porque *s[1]* contiene un carácter del conjunto `"cd"`, concretamente `'c'`, buscando desde el principio.
- *i[3]* vale 5 porque *s[5]* contiene un carácter del conjunto `"cd"`, concretamente `'d'`, y se empieza a buscar desde el final.
- *i[4]* vale 0 porque *s[0]* es el primer elemento que no es ni `'c'` ni `'d'`.
- *i[5]* vale 6 porque *s[6]* es el último elemento que no es ni `'c'` ni `'d'`.
- *i[6]* vale *string::npos* porque `'h'` no se encuentra en *s*.

2.3.4. Vectores de alto nivel: *vector*

Un *vector* de la biblioteca estándar de C++ es un tipo de datos de los conocidos como *contenedores*, pues sirven para contener otros objetos. Todos los contenedores comparten una serie de características y operaciones, lo que simplifica algo su estudio, que se verá en el capítulo 7. Además es un contenedor del tipo *secuencia*.

El tipo *vector* intenta comportarse en su forma más simple como los vectores de bajo nivel. Para ilustrar su empleo más básico, supongamos que queremos trabajar con un listín telefónico. Con los conocimientos que se han adquirido hasta ahora, lo más adecuado parece un vector de estructuras que mantenga cada una un par nombre-número.

```
const int limite = 1000;

struct Entrada {
  string nombre;
  int numero;
};

Entrada listin[limite];

void mostrar_entrada(int i)
{
  cout << listin[i].nombre << '\t' << listin[i].numero << endl;
}
```

Los vectores de bajo nivel deben tener un tamaño conocido en tiempo de compilación. Si no se acierta con el valor adecuado de *limite*, sucederá que o bien desperdiciamos memoria o nos será muy costoso conseguir más. También se podría haber creado un pseudo-vector en memoria dinámica, pero agrandar su tamaño requeriría ocuparnos de la gestión de la memoria. En cambio con el tipo *vector*, es mucho más fácil; en el ejemplo sólo habría que cambiar la definición de *listin*:

```
vector<Entrada> listin(limite);
```

Como se observa, el tipo base del *vector* se pasa como parámetro y encerrado entre ángulos, no entre paréntesis. Se dice que *vector* es una *plantilla*, o un tipo *paramétrico* o *genérico*. Esto se estudiará en §4.8. También se ve que el límite se pasa no entre corchetes sino entre paréntesis. En efecto, el número inicial de elementos es un parámetro de la construcción de un objeto de tipo *vector*.

Puede ser frecuente equivocarse al principio con los paréntesis y los corchetes. Obsérvense las dos declaraciones siguientes:

```
vector<int> v1(100);
vector<int> v2[100];
```

Ambas declaraciones son legales: *v1* es un objeto de tipo *vector* de *int*, de tamaño inicial 100, pero *v2* es un vector de bajo nivel de 100 elementos, cada uno de los cuales es un *vector* de *int* vacío, de tamaño inicial 0. Los errores se producirían al intentar utilizar uno de los dos objetos incorrectamente.

Siguiendo con el ejemplo del listín telefónico, la función *mostrar_entrada()* quedaría inalterada, pues el tipo *vector* admite el acceso a los elementos a través del operador de índice `[]`.

Inicialización Un *vector* puede construirse vacío, con un tamaño inicial, o mediante una copia de otro vector o de un rango de una secuencia. Como segundo parámetro tras el tamaño inicial puede darse un valor para cada elemento del *vector*. Si no se da, el valor de cada elemento será el predeterminado para el tipo base (que es 0 en el caso de los tipos básicos).

Ejemplo:

```
vector<Entrada> listin(limite);
vector<int> v1(20, 5);                        // cada elemento vale 5
vector<double>* p = new vector<double>(30); // en memoria dinámica
vector<int> v2(v1);                 // también: vector<int> v2 = v1;
vector<int> v3 = vector<int>(10);  // Bien: 10 conv. a vector<int>
vector<int> v4 = 10;                // ERROR: no hay conv. implícita
char s[] = "Linux Is Not UniX";
vector<char> v5(s, &s[sizeof(s) - 1]); // rango; copia de caracteres
```

Tamaño Un *vector* crece según se necesite, sin más que hacer uso de las operaciones de inserción o añadidura que se verán.

> No obstante, a veces es conveniente manejar el tamaño del *vector* explícitamente. La función miembro *vector::size()* nos da el número de elementos, *vector::empty()* nos dice si el *vector* está vacío (tamaño 0), *vector::capacity()* nos da el tamaño de memoria (en número de elementos) reservada y *vector::max_size()* el tamaño del vector más grande posible, mientras que *vector::resize()* nos permite modificar el tamaño del *vector*. Esta función es análoga a *realloc()* de la biblioteca estándar de C para el manejo de memoria dinámica. El cambio de tamaño de un *vector* puede implicar mover todos sus elementos. Para evitar

esto en lo posible, existe la operación *vector::reserve()*, que hace que los cambios de tamaño no necesiten pedir más memoria hasta que se sobrepase el valor pasado a dicha función miembro.

Acceso a los elementos Como con los *string*s, se accede a un elemento cualquiera de un vector mediante el operador de índice `[]` o la función miembro *vector::at()*. La primera forma es más eficiente pero no comprueba si el índice cae dentro del rango, por lo que debería emplearse sólo cuando se esté seguro de que el índice es correcto. La operación *vector::at()* comprueba los límites, lanzando la excepción estándar *out_of_range* si es preciso. Las excepciones se estudiarán en el capítulo 5.

Además de esto, se dispone de las funciones miembro *vector::front()* para referirnos al primer elemento del *vector*, y *vector::back()* para el último.

Ejemplo:

```
typedef vector<int>::size_type indice;

void fu(vector<int>& v, indice i1, indice i2)
try {
  for (indice i = 0; i < v.size(); i++) {
    // i está con seguridad dentro del rango
    v[i] =             // ... esto es seguro
  }
  v.at(i1) = v.at(i2); // se comprueba el rango

  // ...
}
catch(out_of_range) {
  // ... error de rango
}
```

vector<int>::size_type es un nombre de tipo definido en la biblioteca (en `<vector>`) correspondiente a un tipo entero sin signo que puede representar cualquier valor no negativo válido como índice de un *vector<int>*. En el ejemplo le aplicamos un `typedef` para abreviar la escritura.

Asignación y copia Dos *vector*es del mismo tipo pueden copiarse y asignarse. Si un vector tiene muchos elementos o éstos son de un tipo complejo, esto puede ser una operación costosa, por lo que lo normal y recomendable es pasarlos por referencia.

EJEMPLO:

```
void f1(vector<int>&);       // estilo normal
void f2(const vector<int>&); // estilo normal
void f3(vector<int>);        // estilo raro

void h()
{
  vector<int> v(10000);

  // ...

  f1(v); // paso por referencia; seguramente v se modificará.
  f2(v); // paso por referencia; v no puede modificarse.
  f3(v); // paso por valor: copia de 10000 int's.
}
```

Para poder hacer asignaciones donde se necesite pasar más de un parámetro, se proporciona la función miembro *vector::assign()*.

EJEMPLO:

```
void f(vector<char>& vc)
{
  char s[] = "literal";
  vc.assign(s, &s[sizeof(s) - 1]); // vc == "literal" (sin '\0')
  vector<char> nv;
  nv = vc;  // asignación simple
}
```

La asignación cambia los contenidos de un *vector*. Los elementos que tuviera se destruyen y se sustituyen por copias de los del *vector* que se asigna. Del mismo modo, el tamaño se ajusta automáticamente al nuevo número de elementos copiados.

Observe que dos vectores de bajo nivel no pueden asignarse de esta forma, sino copiando los elementos uno a uno, quizás mediante una función de biblioteca como *memcpy()* o *memmove()*. Esto es así puesto que el nombre de un vector de bajo nivel empleado en una expresión equivale a su dirección de comienzo. Tenga en cuenta siempre estas diferencias:

```
int v[100], w[100] = { 0 };    // todos los elementos de w a 0
vector<int> V(100), W(100, 0); // todos los elementos de W a 0
```

```
v = w; // ERROR. v es &v[0], no es un valor-i.
V = W; // Bien. V _*NO*_ es &V[0].
```

Comparación Dos *vector*es pueden compararse empleando los operadores habituales ==, <, y los demás. De nuevo, observe la gran diferencia con los vectores de bajo nivel: con ellos, la comparación mediante estos operadores es sintácticamente correcta, pero lo que se comparan son las direcciones de comienzo de los vectores, que obviamente siempre serán distintas a no ser que se compare un vector consigo mismo. Siguiendo con las declaraciones anteriores:

```
if (v == w) // ... compara &v[0] con &w[0]: siempre false

if (V == W) // ... Bien, compara los contenidos de V y W
```

Otras operaciones Los *vector*es, como secuencias que son, pueden ser tratados como tales, y sobre ellos se pueden efectuar operaciones propias de *listas* y de *pilas*.

Si tratamos o consideramos a un *vector* como una lista podemos *insertar* elementos con *vector::insert()* y *quitarlos* con *vector::erase()*.

Si lo tratamos o consideramos como una pila podemos añadir (*apilar*) elementos con *vector::push_back()* y quitar el último (*desapilar*) mediante la operación *vector::pop_back()*.

Por último, otra operación que merece la pena añadir es *vector::swap()*, que intercambia eficientemente dos *vector*es.

Otros tipos relacionados El tipo *vector* recién estudiado está diseñado como un mecanismo general para contener valores, pero no admite operaciones matemáticas sobre vectores. Podrían habérsele añadido con relativa facilidad, pero su generalidad y flexibilidad hubieran hecho imposible optimizaciones consideradas cruciales en trabajos serios de Matemáticas. Consecuentemente la biblioteca estándar de C++ proporciona otro tipo vector menos general y flexible, pero diseñado específicamente para computación numérica. Se llama *valarray* y se define en `<valarray>`.

Ejemplo:

```
#include <valarray>
const double pi = 3.14159265358;
```

```
void f(valarray<double>& a1, valarray<double>& a2)
{
  valarray<double> a = a1 * pi + a2 / a1;
  a2 += a1 * pi;
  a = abs(a);
  double d = a2[7];
  // ...
}
```

Para representar un conjunto de valores lógicos o booleanos se puede emplear la especialización *vector<bool>*, pero la mayoría de las veces habría que considerar la utilización en su lugar del tipo *bitset* (definido en `<bitset>`).

2.4. Operadores

La mayoría de lenguajes de programación modernos tienen la capacidad de crear subrutinas o subprogramas. En C++ se llaman *funciones*. Todas devuelven algo, tienen un valor de retorno, aunque este valor sea «nada», lo que se expresa con la palabra reservada `void`, que equivale aquí a un tipo de datos[5].

Los operadores son símbolos que expresan operaciones que se aplican a uno, dos o tres operandos, y devuelven un valor. Los operadores son como funciones. Sólo hay dos diferencias entre el empleo de un operador y una llamada a función:

- La sintaxis es diferente. Un operador se *llama* poniéndolo normalmente entre sus operandos (*parámetros*), si es binario o ternario; y si es unario se pone inmediatamente detrás o delante.
- Al emplear un operador, el compilador determina automáticamente a qué *función* llamar. Por ejemplo; si se utiliza el operador `+` con parámetros de coma flotante, el compilador «llama» a la función que realiza la adición de números en coma flotante (aunque esta «llamada» seguramente será la acción de insertar código en línea, o una instrucción del coprocesador matemático). Pero si se utiliza el mismo operador `+` con un número entero y otro en coma flotante, el compilador «llama» a una función especial que convierte el entero a flotante y luego «llama» al código que suma dos números en coma flotante.

[5]Este caso es el del *procedimiento* (*procedure*) de Pascal. No se vio que valiera la pena un nuevo nombre en C ni C++ para ello.

Es importante darse cuenta de que los operadores son simplemente una forma diferente de llamada a función, pues en C++ uno puede definir qué función llamará el compilador cuando encuentre operadores utilizados con tipos definidos por el usuario. Esto se llama *sobrecarga* (o redefinición) de operadores, y se verá más adelante.

Según la posición del operador frente a los operandos, un operador puede ser *prefijo* (si va delante), *infijo* (si va en medio) o *postfijo* (si va detrás).

Aparte de esto hay tres características adicionales que deben conocerse de un operador:

Aridad Es el número de operandos sobre los que actúa. En C++ existen operadores

Unarios También llamados *monarios*. Actúan sobre un solo operando.

Binarios Actúan sobre dos operandos.

Ternario Actúa sobre tres operandos. Sólo hay uno: el operador condicional.

Asociatividad Determina en qué orden se *asocian* los operandos del mismo tipo en ausencia de paréntesis. Puede ser:

Por la derecha Tienen esta asociatividad los operadores monarios, el ternario y los de asignación.

Por la izquierda Tienen esta asociatividad los operadores binarios.

Sin asociatividad Para algunos operadores la asociatividad no tiene sentido.

Ejemplo:

`a = b = c` equivale a `a = (b = c)`: `=` es asociativo por la derecha.

`a + b + c` significa `(a + b) + c`: `+` es asociativo por la izquierda.

`sizeof` no tiene asociatividad.

Precedencia Indica la prioridad del operador respecto de otros a la hora de calcular el valor de una expresión.

Los paréntesis sirven para alterar la asociatividad y la precedencia. Póngalos siempre que tenga dudas.

2.4.1. Cuadro resumen de operadores

En la tabla 2.2 se muestran todos los operadores de C++; muchos de ellos no podemos estudiarlos aún, pues sólo tendrán sentido dentro de las *clases*, por ejemplo. De todas formas, se recogen aquí para futuras referencias.

Los operadores se muestran en orden descendente de precedencia: primero los de mayor precedencia, y en último lugar los de menos. Todos los operadores dentro de una misma «caja», delimitada por rayas horizontales, tienen la misma precedencia. La asociatividad se muestra en la segunda columna (**A.**) y se representa por las letras

I	de izquierda a derecha, asociativo por la izquierda
D	de derecha a izquierda, asociativo por la derecha
N	sin asociatividad, no asociativo

Cuadro resumen de operadores			
Operador	**A.**	**Descripción**	**Sintaxis**
`::`	N	resolución de ámbito	*clase* `::` *miembro*
`::`	N	resolución de nombre	*esp_de_nombres* `::` *miembro*
`::`	N	res. de ámbito global	`::` *nombre*
`.`	I	selección de miembro	*objeto* `.` *miembro*
`->`	I	selección de miembro	*puntero* `->` *miembro*
`[]`	I	subíndice	*puntero* `[` *expr* `]`
`()`	I	llamada a función	*expr* `(` *lista-expr* `)`
`()`	N	construcción de valor	*tipo* `(` *lista-expr* `)`
`++`	N	post-incremento	*valor-i* `++`
`--`	N	post-decremento	*valor-i* `--`
`typeid`	N	identificación de tipo	`typeid (` *tipo-o-expr* `)`
`dynamic_cast`	N	conversión dinámica	`dynamic_cast <` *tipo* `> (` *expr* `)`
`static_cast`	N	conversión estática	`static_cast <` *tipo* `> (` *expr* `)`
`reinterpret_cast`	N	conv. no comprobada	`reinterpret_cast<`*tipo*`>(`*expr*`)`
`const_cast`	N	conversión de const.	`const_cast <` *tipo* `> (` *expr* `)`
`sizeof`	N	tamaño de objeto	`sizeof` *expr*
`sizeof`	N	tamaño de tipo	`sizeof (` *tipo* `)`
`++`	N	pre-incremento	`++` *valor-i*
`--`	N	pre-decremento	`--` *valor-i*
`~`	N	complemento	`~` *expr*
`!`	N	negación, ¬	`!` *expr*
`-`	N	cambio de signo	`-` *expr*
`+`	N	signo positivo	`+` *expr*
`&`	N	dirección	`&` *valor-i*
`*`	N	indirección, contenido	`*` *expr*
`new`	N	crear en mem. dinám.	`new` *tipo*
`new`	N	ídem e inicializar	`new` *tipo* `(` *lista-expr* `)`
`new`	N	crear en un lugar	`new (` *lista-expr* `)` *tipo*
`new`	N	ídem e inicializar	`new(`*lista-expr*`)`*tipo*`(`*lista-expr*`)`
`delete`	N	liberar memoria	`delete` *puntero*
`delete[]`	N	liberar vector	`delete[]` *puntero*
`()`	D	modelado de tipo	`(` *tipo* `)` *expr*
`.*`	I	selección de miembro	*objeto* `.*` *puntero-a-miembro*
`->*`	I	selección de miembro	*puntero* `->*` *puntero-a-miembro*
`*`	I	multiplicación	*expr* `*` *expr*
`/`	I	división, cociente	*expr* `/` *expr*
`%`	I	módulo, resto	*expr* `%` *expr*
`+`	I	suma, adición	*expr* `+` *expr*
`-`	I	resta, sustracción	*expr* − *expr*
`<<`	I	bits a la izquierda	*expr* `<<` *expr*
`>>`	I	bits a la derecha	*expr* `>>` *expr*
`<`	I	menor que	*expr* `<` *expr*
`<=`	I	menor o igual que	*expr* `<=` *expr*
`>`	I	mayor que	*expr* `>` *expr*
`>=`	I	mayor o igual que	*expr* `>=` *expr*

continúa en la página siguiente

viene de la página anterior			
Operador	**A.**	**Descripción**	**Sintaxis**
`==`	I	igual que	*expr* `==` *expr*
`!=`	I	distinto de	*expr* `!=` *expr*
`&`	I	conjunción de bits	*expr* `&` *expr*
`^`	I	suma de bits exclusiva	*expr* `^` *expr*
`|`	I	suma de bits	*expr* `|` *expr*
`&&`	I	conjunción, $\wedge$	*expr* `&&` *expr*
`||`	I	disyunción, $\vee$	*expr* `||` *expr*
`?:`	D	expresión condicional	*expr* `?` *expr* `:` *expr*
`=`	D	asignación simple	*valor-i* `=` *expr*
`*=`	D	multiplicación y asignación	*valor-i* `*=` *expr*
`/=`	D	división y asignación	*valor-i* `/=` *expr*
`%=`	D	módulo y asignación	*valor-i* `%=` *expr*
`+=`	D	suma y asignación	*valor-i* `+=` *expr*
`-=`	D	resta y asignación	*valor-i* `-=` *expr*
`<<=`	D	$<<$ de bits y asignación	*valor-i* `<<=` *expr*
`>>=`	D	$>>$ de bits y asignación	*valor-i* `>>=` *expr*
`&=`	D	AND de bits y asignación	*valor-i* `&=` *expr*
`|=`	D	OR de bits y asignación	*valor-i* `|=` *expr*
`^=`	D	XOR de bits y asignación	*valor-i* `^=` *expr*
`throw`	N	lanzamiento de excepción	`throw` *expr*
`,`	I	secuencia (*coma*)	*expr* `,` *expr*

Cuadro 2.2: Resumen de operadores

Operadores explícitos

Son palabras reservadas para los operadores lógicos y de bits. Corresponden en C ISO a macros definidas en la cabecera `<iso646.h>`. Se introdujeron para paliar los problemas de los programadores no americanos con teclados o conjuntos de caracteres sin los signos `&`, `|`, etc. Anteriormente estaban forzados a emplear los horribles *trígrafos*, inventados para C; posteriormente el comité de estandarización ISO, en 1994, definió las macros antedichas para C. C++ las adoptó pero como palabras reservadas del lenguaje. Se muestran en la tabla 2.3.

> En C++, aunque los trígrafos están disponibles por compatibilidad, se alivió un poco su fealdad con los *dígrafos*, y sobre todo con las macros mencionadas, que se convirtieron en palabras reservadas. Hay quien las emplea sin tener otra razón que el gusto o la claridad; normalmente los antiguos programadores de C o de otros lenguajes similares las evitan, pues ya están más acostumbrados a los signos. Para referencia, en la tabla 2.4 se muestran los dígrafos y trígrafos.

and	&&
or	||
not	!
not_eq	!=
bitand	&
and_eq	&=
bitor	|
or_eq	|=
xor	^
xor_eq	^=
compl	~

Cuadro 2.3: Palabras reservadas alternativas

Dígrafos		Trígrafos	
<%	{	??=	#
%>	}	??(	[
<:	[	??<	{
:>	]	??/	\
%:	#	??)	]
%:%:	##	??>	}
		??'	^
		??!	|
		??-	~
		???	?

Cuadro 2.4: Dígrafos y trígrafos

2.4.2. Operadores de asignación

La asignación se realiza mediante el operador `=`, también empleado para la inicialización, que es una asignación inicial cuando se define un objeto. El operador *igual* significa «toma el valor del operando a mi derecha (a menudo llamado *valor-d*[6]) y cópialo a la dirección de memoria especificada por el objeto que hay a mi izquierda (que se suele llamar, así mismo, *valor-i*[7]).

Un valor-d puede ser cualquier constante, variable o expresión que pueda producir un valor, pero un valor-i debe ser una variable o algo que especifique un espacio físico en memoria donde se pueda almacenar algo. Por ejemplo, se puede asignar una constante a una variable (i.e. `a = 4;`), pero no se puede asignar nada a un valor constante: no es un valor-i (`4 = a;`). Por eso, también por ejemplo, no se puede asignar nada a un vector o cadena de bajo nivel: el nombre de tales objetos es su dirección de comienzo, y eso no es un valor-i.

En resumen, un valor-i es cualquier valor que pueda ser fuente de una operación, destino de ella, y se le pueda tomar la dirección. A este respecto, un campo de bits y una variable `register` también son valores-i aunque no cumplan esta última condición.

Ejemplo:

Dadas las definiciones:

```
char* p;
int a[20];
int i;
const double pi = 3.14;
```

algunos ejemplos de operandos que son valores-i son:

p	Un objeto de tipo puntero
i	Un objeto de tipo entero
$*p$	El objeto apuntado por p, un carácter
$a[i]$	El objeto de posición i, un entero

pero no

[6] En inglés, *r-value* o *rvalue*.
[7] En inglés, *l-value* o *lvalue*.

a	Un vector, no define ningún objeto, es `&a[0]`
pi	Un objeto constante, de lectura exclusiva
`p + 2`	Una expresión, no referencia ningún objeto
3	Una constante literal
`'a'`	Otra constante
`"agua"`	Otra más; se refiere a la dirección de comienzo del literal
`i++`	Una expresión
`i = 2`	Otra expresión

Podemos asignar un valor-d a un valor-i, evidentemente, pero no al revés.

C++, al igual que C, introduce una notación nueva para efectuar algunas operaciones a la par que una asignación. Para ello se combina el operador binario en cuestión con el *igual*. Por ejemplo, para añadir 4 a la variable entera x, asignando el resultado a la propia x, se pone `x += 4` en lugar de `x = x + 4` (que también es válido). En la tabla 2.2 se muestran los operadores válidos de asignación combinados.

2.4.3. Operadores matemáticos

Se pueden aplicar a cualquier tipo aritmético (caracteres, enteros, coma flotante) y son los mismos que en cualquier otro lenguaje de programación: *más* para la suma o adición (`+`), *menos* para la resta o sustracción ($-$), *asterisco* para la multiplicación o producto (`*`), *barra* para la división o cociente (`/`) y *porcentaje* para el módulo o resto (`%`). Además, C++ tiene los operadores de incremento y decremento, que se ven en §2.4.4.

Todos estos operadores se combinan con la asignación.

El operador `/` da el cociente de la división entera cuando se aplica a enteros; para coma flotante, da la división real.

El operador `%`, que da el módulo o resto de la división entera, sólo se puede aplicar a enteros, no a números en coma flotante.

El operador `+` también se puede aplicar a la suma de puntero (que apunte a un vector) y entero (§2.3.2).

El operador `-` también se puede aplicar a dos punteros (que apunten al mismo vector) (§2.3.2).

Los operadores `+` y `-` también son unarios, significando `-` el cambio de signo; `+` se introdujo en C ANSI por simetría, y no tiene efecto ninguno más que el estético.

2.4.4. Incremento y decremento

Son unarios, y los únicos operadores aparte de los de asignación que tienen efectos colaterales, pues aparte de devolver un valor modifican su operando. El operador de incremento (`++`) aumenta el valor de su operando (que debe ser un valor-i) en una «unidad», y el de decremento (`--`) lo disminuye. «Unidad» puede tener un significado diferente dependiendo del tipo de datos, sobre todo con punteros (§2.3.2).

El valor producido depende de si el operador se emplea como prefijo o como sufijo (esto es, antes o después de la variable). Como prefijo, el operador cambia el valor de la variable y devuelve este nuevo valor. Como sufijo, el resultado de la expresión es el valor de la variable, y luego ésta se modifica.

2.4.5. Operadores relacionales

Establecen una relación entre los valores de los operandos. Producen un resultado lógico `true` (que equivale al entero 1) si la relación es verdadera, y `false` (que equivale al entero 0) si es falsa. Estos operadores son el *menor* (`<`), el *mayor* (`>`), el *menor-igual* o (`<=`), el *mayor-igual* (`>=`), el *igual-igual* (`==`) y el «distinto» (`!=`), que también se puede expresar con la palabra reservada `not_eq`, como se ve en la tabla 2.3.

Estos operadores se pueden emplear con todos los tipos básicos del C++ y se les puede dar significados especiales (es decir, se pueden redefinir o sobrecargar) para tipos definidos por el usuario.

Hay que tener presente que la comparación de un número en coma flotante con 0 es muy estricta; las comparaciones de números en coma flotante pueden dar sorpresas.

2.4.6. Operadores lógicos

Los operadores lógicos *Y* o conjunción lógica (`&&` o `and`), y *O* o disyunción lógica (`||` u `or`) producen un valor lógico verdadero (`true`) o falso (`false`) basado en la relación lógica de sus operandos. Ambos evalúan «en corto-circuito», como se explicará en el capítulo sobre estructuras de control; es decir, se para la evaluación de la expresión en cuanto puede determinarse la veracidad o falsedad de ésta. Cualquier valor aritmético o puntero distinto de 0 se convierte a `true` y cualquier 0 aritmético o puntero se convierte a `false`.

Los operadores anteriores son binarios. El operador lógico unario *NO* (`!` o `not`) da el valor lógico contrario al de su operando: si éste es `true`, produce

`false`, y viceversa.

Todos estos operadores se pueden aplicar a cualquier tipo o expresión aritmética o puntero.

2.4.7. Operadores de bits

Permiten manipular bits individuales en un número; por lo tanto sólo trabajan sobre operandos enteros o caracteres; preferentemente se emplean operandos `unsigned`. Estos operadores efectúan álgebra booleana sobre cada pareja de bits correspondiente de los operandos para producir el resultado.

El producto binario o *Y* de bits (`&` o `bitand`) produce un 1 como bit de salida si ambos bits de entrada son 1; si no, produce un 0.

La suma binaria (inclusiva) u *O* de bits (`|` o `bitor`) da 1 si cualquiera de los bits de entrada es 1; 0 si ambos bits son 0.

La suma binaria exclusiva de bits o *XOR* (`^` o `xor`) produce un 1 si uno cualquiera de los bits de entrada es 1, pero no ambos, y 0 si ambos bits de entrada son iguales.

Estos operadores son binarios y se pueden combinar con la asignación.

El operador de complemento (a 1) o *NO* binario (`~` o `compl`) es unario y produce el bit opuesto: 1 si el bit de entrada es 0, 0 si el bit de entrada es 1.

> Hay quien incluye en esta categoría de operadores de bits al *menos* unario (`-`), como cambio de signo, pero esto es muy discutible: `-` funciona también sobre reales, y no manipula bits directamente; además su funcionamiento depende de la representación interna de los números.

Los operadores binarios de desplazamiento, que también se pueden combinar con la asignación, también manipulan bits. Son el *menor-menor* (`<<`) y el *mayor-mayor* (`>>`), según el desplazamiento de bits sea a la izquierda o a la derecha respectivamente. Ambos producen como resultado el valor del operando de la izquierda con sus bits desplazados tantas veces como indica el operando de la derecha, que debe ser por tanto una expresión entera positiva. Si el valor de esta expresión es mayor que el número de bits del operando de la izquierda, el resultado es indefinido. Si el operando de la izquierda es `unsigned`, el desplazamiento a la derecha es un desplazamiento *lógico*, de forma que las vacantes que van quedando a la izquierda se van rellenando con ceros; si el operando de la izquierda en cambio es `signed`, el desplazamiento puede ser lógico o *aritmético*; dependiendo del signo, los huecos pueden rellenarse con ceros o con unos.

Estos operadores, como se ve, producen desplazamientos, no rotaciones. No hay operadores en C++ para las rotaciones; incluso aunque es frecuente que haya una instrucción de rotación en ensamblador, es muy fácil construirse funciones de rotación basándose en los desplazamientos, conque Dennis Ritchie y colegas, autores del lenguaje C, consideraron justificado no incluir operadores de rotación, siguiendo el espíritu de construir un lenguaje lo más pequeño posible. Aunque C++ no es precisamente el lenguaje más pequeño posible, Bjarne Stroustrup, su inventor, dejó los operadores de bits tal cual los encontró en C.

Como ejemplo, se muestran las funciones que realizan la rotación de bits. Antes de verlas, procure hacerlas como ejercicio.

```
1  /* Efectúan rotaciones de bits a izquierda y derecha
2     para caracteres de 8 bits
3  */
4
5  unsigned char rol(unsigned char val)
6  {
7    int msb = (val & 0x80) >> 7;  /* bit más significativo */
8    val <<= 1;                    /* lsb == 0              */
9    return val | msb;             /* pone el msb en el lsb */
10 }
11
12 unsigned char ror(unsigned char val)
13 {
14   int lsb = val & 1;            /* bit menos significativo */
15   val >>= 1;                    /* msb == 0                */
16   return val | (lsb << 7);      /* pone el lsb en el msb   */
17 }
```

Los operadores de bits son por lo general extremadamente eficientes porque se traducen directamente a instrucciones de ensamblador. Algunas veces una sola instrucción de C o C++ generará una sola línea de código en ensamblador.

2.4.8. Código en ensamblador

Aunque esto no es un operador, conviene hablar de ello y éste puede ser el lugar más conveniente. La palabra reservada `asm` es un mecanismo de escape que permite escribir código en ensamblador dentro de un programa C++. A menudo se pueden referenciar variables de C++ dentro del código en ensamblador. Curiosamente, en C estándar no existe esta palabra, aunque la mayoría de los compiladores la admiten. Así, el programador puede en

ocasiones, a la hora de escribir el código de más bajo nivel, hacer uso de instrucciones del procesador por ejemplo, mediante la combinación de la compilación condicional del preprocesador y el ensamblador.

Después de la palabra reservada `asm` se pone una cadena de caracteres entre paréntesis. Obviamente, el contenido de esa cadena depende del sistema, aunque se supone que es código en ensamblador.

Ejemplo:

rotaciones.cpp

```
1  /* Ejemplo del uso del ensamblador en C++
2   * con rotaciones de bits a izquierda (roll)
3   * y derecha (rorl).
4   * Estas funciones devuelven un int, pero no hay
5   * ningún return porque el ensamblador se encarga
6   * de todo.
7   * Observe que puede poner cada línea de ensamblador en una
8   * instrucción asm o bien poner una sola instrucción asm con
9   * cada línea en una gran cadena de caracteres con los deli-
10  * mitadores propios del ensamblador, en este caso el salto
11  * de línea y el tabulador.
12  */
13
14 /* Esto está escrito sólo para el ensamblador de GNU (gas) en
15  * un Linux con GCC y para IA32. Así que nos ahorraremos muchos
16  * mensajes de error si intentamos compilar en otro sistema con
17  * lo siguiente:
18  */
19 #if !defined __linux || !defined __x86 || !defined __GNUC__
20 #error Error: sólo compila en Linux para x86 con GCC.
21 #endif
22
23 int roll(int a, int n)
24 {
25   asm("movl 12(%ebp),%ecx"); // ecx = n
26   asm("movl 8(%ebp),%eax");  // eax = a
27   asm("roll %cl,%eax");      // rota eax, cl bits a la izda.
28 }
29
30 int rorl(int a, int n)
31 {
32   asm("movl 12(%ebp),%ecx\n\t" // ecx = n
33       "movl 8(%ebp),%eax\n\t"  // eax = a
34       "rorl %cl,%eax");        // rota eax, cl bits a la dcha.
35 }
36
```

```
37 #ifdef TEST  /* PRUEBA */
38 #include <iostream>
39 #include <iomanip>
40 using namespace std;
41
42 int main()
43 {
44   cout << "Introduzca un número en base 16, 8 ó 10: ";
45   int a;
46   cin >> setbase(0) >> a;
47   cout << hex << setiosflags(ios::showbase)
48        << "roll(" << a << ", 1) = " << roll(a, 1) << endl
49        << "rorl(" << a << ", 1) = " << rorl(a, 1) << endl;
50 }
51 #endif
```

2.4.9. Operador condicional

Es el único ternario que tiene el lenguaje. No se puede redefinir o sobrecargar. Es realmente un operador puesto que produce un valor, a diferencia de la instrucción similar `if-else`. Consiste en tres expresiones. Si la primera se evalúa a `true`, la segunda, que va detrás del signo de interrogación (?), se evalúa y devuelve, convirtiéndose su valor en el devuelto por el operador. Si en cambio la primera expresión es falsa (`false`), es la tercera y última expresión, que va detrás del signo de dos puntos (:), la que se evalúa y devuelve como resultado del operador.

Este operador se puede emplear por sus efectos colaterales sólo (despreciando el valor devuelto, y en este caso es análogo a un `if-else`, sólo que más compacto) o por el valor que produce.

Ejemplo:

```
a = --b ? b : (b = -99);
```

Supuesto que a y b sean variables enteras, la instrucción anterior asigna a a el valor de b si el resultado de decrementar b es distinto de 0. Si b, tras decrementarlo, valiera 0, entonces tanto a como b acaban valiendo -99. Observe que b siempre se decrementa, pero acaba valiendo -99 sólo si ese decremento hace que valga 0.

Si sólo nos interesara el efecto producido en b, podríamos haber desechado el resultado[8]:

```
--b ? b : (b = -99);
```

que es equivalente a la instrucción siguiente.

```
if (--b == 0) b = -99;
```

En este caso no hace falta la rama `else` puesto que no se hace nada. De hecho, en la expresión condicional podría haberse sustituido la segunda expresión, `b`, por una constante, como `0` por ejemplo, para que el código fuera un poquito más rápido al no tener que evaluar la variable b[9].

2.4.10. Operadores de dirección e indirección

Son los siguientes:

`*`	Indirección, desreferencia o contenido
`&`	Dirección o referencia
`->`	Puntero selector de miembro
`.`	Selector de miembro
`[]`	Subíndice
`->*`	Puntero selector de puntero a miembro
`.*`	Selector de puntero a miembro

Todos estos operadores se vieron en §2.3.2.

2.4.11. Operador de tamaño

El operador de tamaño o `sizeof` es especial en cuanto que se evalúa en tiempo de compilación. Da información sobre la cantidad de memoria ocupada por un objeto o la necesaria para reservar un objeto de un tipo de datos determinado. Su operando puede ser por tanto una expresión (normalmente una variable) o bien el nombre de un tipo de datos entre paréntesis.

[8] Esto se llama «conversión *de línea*»: el tipo de la expresión «es convertido» implícitamente por el compilador a *void*; se puede poner explícitamente con el operador de conversión, pero no hace falta: `(void)(--b ? b ? (b = -99));`.

[9] Algunos compiladores, como el de GNU, permiten omitir la segunda expresión: `--b ? : (b = -99);` esto no es estándar.

El tamaño lo da en bytes, y el tipo del número devuelto es un entero sin signo dependiente del sistema; se define con el nombre *size_t* en la cabecera `<stddef.h>` o `<cstddef>` y otras muchas. Este tipo se utiliza en muchas otras ocasiones en la biblioteca estándar.

Ejemplo:

Como se ha dicho, el operador se evalúa en tiempo de compilación: el compilador sustituye la expresión con `sizeof` por el valor constante resultante. Por lo tanto el siguiente fichero no se puede compilar:

```
1 #include <cstddef>  // Por size_t
2
3 extern int v[];  // Declaración. Definición en otro sitio
4
5 size_t f()
6 {
7   return sizeof v; // ERROR. Tipo incompleto
8 }
```

El error que da el compilador de GNU es:

```
'sizeof' applied to incomplete type 'int[]'
```

O sea: «`sizeof` aplicado al tipo incompleto *int[]*». En efecto, el compilador no puede determinar el tamaño del vector v, puesto que ese tamaño se especifica en otro fichero, donde se defina. La solución en este caso podría consistir en dar el tamaño en la declaración, teniendo cuidado de que coincidiera con la de la definición.

De este ejemplo vemos también que cuando el operando de `sizeof` es un vector o cadena de bajo nivel, el nombre del vector o cadena no se convierte en la dirección de comienzo, o sea en este caso, `&v[0]`. Si así fuese, `sizeof v` daría siempre el tamaño de un puntero, puesto que una dirección es un valor de tipo puntero. En cambio, aplicado a un vector o cadena, da el tamaño total en bytes de dicho vector o cadena (no el número de elementos, a menos que éstos sean *char* obviamente, de tamaño 1 byte). Cuando se aplica `sizeof` a una referencia, nos da el tamaño del objeto referenciado.

2.4.12. Operador secuencia o de evaluación

El operador de secuencia, de evaluación o *coma* (`,`) sirve para evaluar distintas expresiones seguidas, de izquierda a derecha, devolviendo como valor

de la operación el resultado de la expresión de más a la derecha. Las demás expresiones sirven solamente por sus efectos colaterales, pues sus resultados se pierden.

Ejemplo:

```
a = b++, c++;   // Cuidado. Significa: (a = b++), c++;
a = (b++, c++); // Bien, significa: b++; a = c; c++;
int i = (1, 2); // absurdo, pero i vale 2.
int v = (srand(0), rand());
for (i = 0, j = 0; /* ... */; i++, j++)
  // ...
```

Como se ve, en la primera línea, por no poner paréntesis, el significado es «asignar b a a, incrementar b; se devuelve como resultado el valor de c, que se desecha, y por último se incrementa c». En cambio, como se ve en la segunda línea, lo más lógico es esto: «incrementar b, devolver el valor de c y asignarlo a a, incrementar c».

La cuarta línea equivaldría a

```
srand(0);
int v = rand();
```

Éstas son funciones de la biblioteca estándar de C, declaradas en `<cstdlib>`, de números pseudo-aleatorios.

El último ejemplo es el de uso más común de este operador: en un bucle `for` con varias variables de control.

No se puede utilizar este operador cuando el compilador espera una expresión constante, como en

```
int v[2, 3];  // ERROR
```

2.4.13. Operadores de conversión

Los tipos aritméticos pueden mezclarse libremente en expresiones y asignaciones; el compilador automáticamente hará las conversiones convenientes. Una conversión de un tipo de menor tamaño a uno relacionado de mayor tamaño, donde por tanto no se pierde información, se llama *promoción*. Ejemplo: de *char* a *int*. En otros casos, puede que se pierda información.

EJEMPLO:

```
int a = 456;
unsigned char b = a; // Cuidado: ¿cuánto vale 'b'?
a = 'a';             // Bien: promoción char -> int
```

Las conversiones, salvo las promociones, muchas veces son peligrosas y fuente de posibles errores. Cuando el compilador no es capaz de efectuar la conversión que queremos por sí mismo, el programador puede hacerla explícita mediante los operadores que vamos a ver.

Operador de modelado Se mantiene por compatibilidad con el lenguaje C, y es el único que existe en dicho lenguaje. Se llama el operador de modelado, o *molde*, y consiste en poner delante de la expresión a convertir el tipo deseado entre paréntesis:

EJEMPLO:

```
int i = 5, j = 2;
double x = i / j;    // truncamiento. x == 2.0

x = (double)(i / j); // Cuidado: x == (double) 2 == 2.0
x = (double)i / j;   // Bien. x == 2.5. i se convierte a double
x = i / (double)j;   // Bien, ahora j se convierte a double
x = (double)i / (double)j; // Bien, más explícito.
```

Operador de construcción de tipo Es en realidad la construcción de un tipo de datos. La sintaxis es la misma que la de los *constructores*, que se verán en el capítulo de las clases. Ahora lo que va entre paréntesis es la expresión a convertir, tras el nombre del tipo, como si éste fuera una función. Por eso a veces esta notación se llama *notación funcional*.

EJEMPLO:

```
enum opcion { A, B, C };
opcion a = (opcion)2;  // notación de C
opcion b = opcion(2);  // notación funcional (construcción)
```

Aunque esta notación está pensada principalmente para la conversión y construcción de tipos definidos por el usuario, se amplió a los tipos básicos, que en ocasiones se comportan *como si* fueran de aquella clase. Así, se puede poner lo siguiente, por ejemplo:

```
int i = 5;      // Bien, como siempre
int j(5);       // el mismo significado
int k = int(5); // igual, pero más explícito
int m = int();  // m = 0
```

El nombre del tipo debe ser una sola palabra: no puede aplicarse esta notación a un tipo como *char**, por ejemplo. Pero la solución a esto sería emplear un nombre alternativo con `typedef`:

```
void f(char* c, size_t n)
{
  void* p = malloc(n);   // p apunta a n bytes de memoria

  char* c1 = (char*) p;  // notación de C
  char* c2 = char*(p);   // ERROR

  typedef char* pchar;
  char* c2 = pchar(p);   // Bien, ahora sí vale.
}
```

Como se ha dicho, las conversiones son por lo general peligrosas. C++ es un lenguaje fuertemente tipado y además, normalmente no hace falta mencionar una conversión explícitamente con estos operadores. Pero C++ no prohíbe hacer cualquier conversión, por rara que parezca: simplemente evita que se produzca por error. Si el programador quiere hacer algo raro o considerado peligroso, adelante, pero tendrá que escribirlo explícitamente, para evitar que sea por error. De esta manera, el comité ANSI que normalizó el lenguaje decidió que no todas las conversiones eran iguales, y que el programador debería especificar qué clase de conversión quería realmente. A tal efecto, creó cuatro nuevos operadores y palabras reservadas. Para los cuatro, la sintaxis es la misma: la palabra reservada y, entre ángulos, el nombre del tipo al que se quiere convertir la expresión, que va al final entre paréntesis.

Operador `static_cast` El operador `static_cast` convierte entre tipos relacionados, tales como punteros de distinto tipo base, o de enumeraciones o coma flotante a enteros. Algunas conversiones de éstas son perfectamente transportables y lógicas, y el compilador, con este operador, puede efectuar algunas mínimas comprobaciones. El operador de construcción visto anteriormente equivale a este operador para los tipos básicos.

Operador `reinterpret_cast` Este operador maneja conversiones entre tipos no relacionados, como entre punteros y enteros. Éstas son las conversiones más peligrosas; empleando estos operadores, el programador puede buscar en el código fuente las conversiones más fácilmente que con el operador *molde* de C o con el de construcción. El compilador no aplica con este operador ninguna comprobación, todo es responsabilidad del programador. Normalmente el resultado del operador es un patrón de bits idéntico al de la expresión original, pero que ahora se *reinterpreta* como del nuevo tipo. Para convertir entre punteros a funciones distintas éste es el operador que hay que utilizar.

Ejemplo:

```
#include <cstdlib>          // void* malloc(size_t);
#include "Dispositivos.h"   // definición del tipo Disp_ES

void f(size_t n)
{
  int* p = static_cast<int*>(malloc(n * sizeof(int)));
  Disp_ES* dl = reinterpret_cast<Disp_ES*>(0xFF00);
  // ...
}
```

La función *malloc()* de la biblioteca estándar de C reserva tantos bytes de memoria dinámica como indica su parámetro y devuelve la dirección de la zona reservada, como un puntero genérico[10]. El compilador no sabe a qué tipo apunta un puntero genérico, así que el programador tiene que decírselo: en este caso, a entero.

En la siguiente línea, se reinterpreta el entero hexadecimal FF00 como una dirección, donde el programador sabe que está cierto dispositivo de entrada/salida, y se convierte este entero a puntero a un tipo adecuado definido por el usuario.

Operador `const_cast` Este operador, `const_cast`, maneja las conversiones de constantes; es decir, «quita» el atributo de constante a un objeto. Esto también suele ser peligroso (¿para qué, si no, se definió el objeto `const`?), y de ahí este nuevo operador.

[10]Por supuesto, en C++ no hay necesidad de utilizar esta función; para ello están los operadores de memoria dinámica.

Ejemplo:

El siguiente programa ilustra cómo uno puede, si lo hace a propósito, evitar el atributo `const` de una variable.

```
1  // Ejemplo de cómo evitar un const
2
3  #include <iostream>
4  using namespace std;
5
6  void f(const int& pc)
7  {
8    int& p = const_cast<int&>(pc);
9    p = 100;     // modificación de const a través de no-const
10 }
11
12 int main()
13 {
14   int i = 1;
15
16   cout << "Antes:   i = " << i << endl;
17   f(i);
18   cout << "Después: i = " << i << endl;
19 }
```

Operador `dynamic_cast` Este último operador realiza comprobaciones en tiempo de ejecución y se estudiará en el capítulo sobre polimorfismo.

Nótese que el operador de modelado, al estilo C, equivale al `static_cast`, `reinterpret_cast` y `const_cast`. Debe evitarse en beneficio de estos tres; el de construcción se puede y debe emplear algunas veces cuando se quiere dejar explícito que se está construyendo un objeto de un nuevo tipo.

2.4.14. Operadores de memoria dinámica

Se ha visto cómo definir variables en memoria automática o de pila, y en memoria estática. Hay otra área de memoria disponible, y es la memoria dinámica. Básicamente, el programador dispone de una amplia cantidad de memoria que solicita poco a poco para ir almacenando objetos. En ausencia de un «recolector automático de basura», esta memoria permanece ocupada hasta que explícitamente el programador la libere, devolviéndola a la zona de memoria dinámica libre, de donde podrá volver a utilizarla.

En C, no existía ningún operador para la reserva y liberación de memoria dinámica, sino que su manejo se efectuaba a través de las funciones de biblioteca *malloc()*, *calloc()*, *realloc()* y *free()*, declaradas en la cabecera `<stdlib.h>`.

Aunque estas funciones siguen estando disponibles en C++, hay una mejor alternativa: los operadores `new` y `delete`.

Aún no se tienen los elementos de juicio suficientes para apreciar que son imprescindibles en C++ y no sólo una conveniencia sobre las funciones antedichas de C; por ahora, baste decir que `new` se encarga automáticamente de todo lo que el programador tenía que hacer manualmente con *malloc()*: `new` determinará el tamaño de memoria que tiene que reservar, la inicializará con algún valor especificado si es preciso[11], se encargará (con algo de ayuda nuestra) de lo que ocurra si no hay bastante memoria, y devolverá un puntero del tipo apropiado. Y para todo esto, el programador sólo tiene que darle como operando un tipo de datos.

Ejemplo:

Supongamos que estamos implementando una estructura de datos tipo árbol binario; emplearíamos una estructura como

```
struct Nodo {
  void* datos;
  Nodo* izdo;
  Nodo* dcho;
};

Nodo* pnodo;
```

y en algún momento tendremos que crear un nodo nuevo en tiempo de ejecución. Con *malloc()*, como se haría en C, sería así:

```
#include <stdlib.h>  /* malloc(), exit(), EXIT_FAILURE */
#include <stdio.h>   /* fprintf(), stderr, NULL        */

/* ... */

if ((pnodo = (Nodo*)malloc(sizeof Nodo)) == NULL) {
  fprintf(stderr, "ERROR FATAL: sin memoria.\a\n");
  exit(EXIT_FAILURE);
```

[11] Esto es, realmente *construirá* un objeto en memoria dinámica, pues si dicho objeto tiene algún método de inicializarse, `new` lo ejecutará. En otras palabras, llama al *constructor* del objeto, que se verá en el capítulo sobre las *clases*.

```
}
/* seguir... */
```

Pero con `new`, sencillamente:

```
pnodo = new Nodo;
```

Para inicializar un objeto se pasan entre paréntesis los parámetros de la construcción. Para un tipo básico, por ejemplo:

```
int* p = new int(5);
```

crearía un entero en memoria dinámica, le daría el valor inicial 5 y asignaría su dirección al puntero *p*.

La memoria dinámica permanece a lo largo del programa hasta que explícitamente se libera. Para ello se dispone del operador `delete`, que recibe como operando un puntero que obligatoriamente apunte a una zona reservada por `new`.

Ejemplo:

Siguiendo con los dos ejemplos anteriores, para liberar la memoria ocupada por el *Nodo* y el *int*:

```
delete pnodo;
delete p;
```

> Un puntero nulo como operando de `delete` no da error ni tiene ningún otro efecto; es inocuo. Por otra parte, `delete` no cambia su operando puntero (éste sigue apuntando al mismo sitio, contiene la misma dirección; lo que pasa es que esa dirección ya no está reservada, puede volver a emplearse para guardar otro objeto distinto). Por eso, para evitar una destrucción accidental de la misma zona de memoria, lo que sería malo, algunos programadores tienen la costumbre de hacer algo así:
>
> ```
> delete p; p = 0;
> ```
>
> asegurándose de que si por descuido volvieran a hacer `delete p;`, esto sería inofensivo.

Por último, hay que decir que los operadores `new` y `delete` se pueden sobrecargar o redefinir para que obtengan y liberen respectivamente memoria de la forma que uno quiera. Esto puede ser interesante para algunos tipos definidos por el usuario o para implementar políticas de gestión de memoria; por ejemplo, un *recolector de basura*.

Vectores dinámicos

Es posible reservar memoria para una ristra de objetos; es decir, para un vector de bajo nivel. La sintaxis de `new` es igual, sólo que ahora el tipo es un vector y que no se pueden inicializar sus elementos con un valor específico.

Ejemplo:

```
int* vp = new int[50];
double* const v = new double[10];
```

El modificador `const` de *v* hace que éste se comporte más como un vector «normal», puesto que ahora *v* no sería modificable (esto es, no podríamos hacer por ejemplo `v++` ni `v = ...`); no obstante, sigue siendo un valor-i (por ejemplo, podemos tomar su dirección, `&v`, que *no* sería la de comienzo del vector, y `sizeof v == sizeof(double*)`).

La primera instrucción del ejemplo anterior, por abreviar, hace que se reserve memoria dinámica para almacenar 50 enteros. Sin embargo, solamente hemos obtenido la dirección de comienzo, que guardamos en *vp*; que contendría exactamente lo mismo que si hubiéramos puesto

```
int* vp = new int;
```

para crear un solo objeto. El que ha escrito el código, o lo lee, sabe que *vp* es realmente la dirección de comienzo de un vector de 50 enteros, por lo que tienen sentido expresiones como `vp[2]` o `vp++`, pero ¿cómo destruimos el vector? Si hacemos

```
delete vp;
```

se liberará la memoria del vector entero, pues fue reservada de una sola vez y la rutina de asignación recordará su tamaño, pero sólo se llamará al destructor del primer objeto. El destructor es la rutina que realiza la tarea de limpieza, o cualquier otra especificada, justo antes de destruirse un objeto. Y es que como ya se dijo en la sección 2.3.2, un puntero no contiene información

de si apunta a un solo objeto o a una ristra de ellos. Esto es muy importante sobre todo para objetos definidos por el usuario (clases), por lo que hay un mecanismo para decirle a `delete` que *vp* contiene la dirección de comienzo de un vector:

```
delete []vp;
```

> En las primeras versiones de C++ el programador tenía que especificar entre los corchetes el número de elementos, pero como esto podía fácilmente provocar errores y descuidos y para el compilador el coste de recordar esta cantidad es muy pequeño, se decidió que era mejor escribir el tamaño en un solo sitio. Aunque se permite aún poner el número de elementos a destruir entre los corchetes.

No hay memoria

Cuando se pide memoria dinámica pero no hay, las funciones de C *malloc()*, *calloc()* o *realloc()* devuelven 0, el puntero nulo; esto obliga al programador a comprobar el valor devuelto en cada llamada. ¿Qué ocurre si `new` no encuentra memoria disponible?

En primer lugar intenta llamar a una función especificada por el programador para manejar el caso. Esta función no debe devolver ni recibir nada, y se «instala» como manejador de falta de memoria mediante la función de la biblioteca estándar de C++ *set_new_handler()*, declarada en la cabecera `<new>`, que recibe como parámetro la dirección de nuestra función manejadora.

Ejemplo:

El siguiente programa intenta probar lo recién dicho e ilustra cómo emplear la función anterior.

newhandler.cpp

```
1  // Prueba de un manejador de falta de memoria
2  #include <iostream>  // cerr, operator <<(), endl
3  #include <new>       // set_new_handler()
4  #include <cstdlib>   // exit(), EXIT_FAILURE
5  using namespace std;
6
7  void amnesia()
8  {
9    cerr << "Error fatal: memoria agotada." << endl;
```

```
10    exit(EXIT_FAILURE);
11  }
12
13  int main()
14  {
15    set_new_handler(amnesia);
16    volatile int* p;  // volátil para que el compilador no optimice
17    while (true) p = new int[1024];  // agotará la memoria
18  }
```

Si no se emplea esta técnica, esto es, si no existe ningún manejador instalado, entonces `new` lanza una excepción estándar llamada *bad_alloc*, que podemos capturar y tratar (esto se verá en el capítulo correspondiente). Si no la capturamos, entonces se lanza un manejador de excepción no capturada, que también podrá el programador especificar; si no lo hace, se lanza uno estándar.

En cualquier caso, el comportamiento está bien definido; en cambio, si no comprobamos lo devuelto por *malloc()* y seguimos con el programa, los errores surgirán de forma imprevista en la ejecución y será muy difícil detectarlos.

> Es posible también obtener un comportamiento análogo al de *malloc()*, o sea, que `new` devuelva 0, pasándole el parámetro especial *nothrow*, como en
>
> ```
> int* p = new(nothrow) int[1000];
> ```
>
> Esto no está recomendado y se mantiene solamente por compatibilidad con versiones antiguas de C++.

Sobrecarga de operadores

Es más importante que la de funciones (salvo para los constructores) y su verdadera utilidad se verá cuando se estudien las *clases*. De momento se verá un ejemplo que utilice sólo lo visto hasta ahora.

Ejemplo:

Se va a hacer un pequeño programa que trabaje sobre números complejos.

Para ello se creará una estructura (`struct`) que represente un número complejo, y varias funciones que operarán sobre ellos. Luego se probarán estas funciones.

Empezaremos como se haría en C, sin sobrecarga de operadores.

complejos1.cpp

```
1  // Operaciones básicas sobre complejos sin utilizar sobrecarga
2
3  #include <iostream>
4  using namespace std;
5
6  struct Complejo {
7    double re, im;
8  };
9
10 Complejo crear(double re, double im)
11 {
12   Complejo t = {re, im};
13   return t;
14 }
15
16 void mostrar(const Complejo& z)
17 {
18   cout << '(' << z.re << ", " << z.im << ')' << endl;
19 }
20
21 Complejo sumar(const Complejo& a, const Complejo& b)
22 {
23   Complejo t = { a.re + b.re, a.im + b.im };
24   return t;
25 }
26
27 Complejo restar(const Complejo& a, const Complejo& b)
28 {
29   Complejo t = { a.re - b.re, a.im - b.im};
30   return t;
31 }
32
33 int main()
34 {
35   Complejo a, b, c, d;
36
37   a = crear(1.0, 1.0);
38   b = crear(2.0, 2.0);
39   c = sumar(a, b);
40   d = sumar(b, restar(c, a));        // confuso; d = b + c - a
41
42   cout << "c = ";
43   mostrar(c);
44   cout << "d = ";
45   mostrar(d);
46 }
```

Como se ve, la sintaxis funcional es bastante confusa (para quien no esté acostumbrado, claro; por ejemplo, no para un programador de Lisp). Tratándose de un tipo numérico sería mucho más cómodo poder hacer cosas como

```
d = b + c - a;
```

y esto es lo que se consigue con la sobrecarga de operadores. Veamos la nueva versión:

complejos2.cpp

```
1  // Operaciones básicas sobre complejos utilizando sobrecarga
2
3  #include <iostream>
4  using namespace std;
5
6  struct Complejo {
7    double re, im;
8  };
9
10 Complejo crear(double re, double im)
11 {
12   Complejo t = {re, im};
13   return t;
14 }
15
16 void mostrar(const Complejo& z)
17 {
18   cout << '(' << z.re << ", " << z.im << ')' << endl;
19 }
20
21 Complejo operator +(const Complejo& a, const Complejo& b)
22 {
23   Complejo t = { a.re + b.re, a.im + b.im };
24   return t;
25 }
26
27 Complejo operator -(const Complejo& a, const Complejo& b)
28 {
29   Complejo t = { a.re - b.re, a.im - b.im };
30   return t;
31 }
32
33 int main()
34 {
35   Complejo a, b, c, d;
36
```

```
37    a = crear(1.0, 1.0);
38    b = crear(2.0, 2.0);
39    c = a + b;
40    d = b + c - a;
41
42    cout << "c = ";
43    mostrar(c);
44    cout << "d = ";
45    mostrar(d);
46  }
```

Al sobrecargar o redefinir un operador cambiamos la función que llama el compilador cuando ve el empleo de aquél: ahora llamará a la función escrita por nosotros para ese operador. Esa función se declara y define exactamente como cualquier otra, salvo que el nombre se sustituye por la palabra reservada `operator` seguida del símbolo o nombre del operador. Cuando el compilador, siguiendo el ejemplo de antes, ve la expresión

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

y que *a* y *b* son de tipo *Complejo*, busca la definición del operador `+` y sustituye la expresión por la llamada

```
c = operator +(a, b);
```

que también se puede escribir así, es perfectamente válida en C++, aunque no se hace nunca salvo para propósitos didácticos, como aquí.

Y esto es todo, salvo que hay algunas reglas y restricciones a la hora de sobrecargar operadores. Aquí están:

- La precedencia, asociatividad y sintaxis de los operadores sobrecargados es la misma que la de los originales, no se puede variar.
- No podemos «inventar» operadores nuevos como @ o **.
- Los operadores sobrecargados no pueden tener parámetros prefijados.
- No se puede sobrecargar un operador sólo para tipos básicos del lenguaje. Es decir, no podríamos redefinir la suma de enteros para que multiplique:

  ```
  int operator(int a, int b) { return a * b; } // ERROR
  ```

  Al menos un parámetro debe ser de un tipo definido por el usuario.

- Los operadores sobrecargables son los de C++ exclusivamente; no se pueden sobrecargar los operadores del preprocesador, como `#`, `##` o `defined`.
- No todos los operadores pueden sobrecargarse. No se pueden de ninguna manera `.`, `.*`, `::`, `?:` y `sizeof`. Y los operadores `[]`, `=`, `->` y `(`*tipo*`)` sólo se pueden redefinir dentro de *clases*.
- Los operadores unarios y binarios al mismo tiempo, como `+`, `-` o `&` se pueden sobrecargar en las dos formas; recibirán uno o dos parámetros dependiendo del caso.
- Los operadores `++` y `--` se pueden sobrecargar tanto en su forma prefija como en la sufija. En este último caso reciben un parámetro extra de tipo *int* que no se utiliza y sólo sirve para resolver la sobrecarga.
- Los operadores `new` y `delete` se pueden redefinir con restricciones: sólo se puede cambiar la forma de obtener memoria de `new`; o sea, uno puede definirse su propio asignador de memoria. Además se puede redefinir el `new` global o sólo para una *clase* determinada.

2.5. Expresiones

Una expresión es una secuencia de símbolos que produce un valor. Los símbolos incluyen probablemente operadores y operandos. Una expresión podría estar compuesta por ejemplo por:

- una constante literal.
- un objeto, constante o no.
- una llamada a función.
- una combinación de los anteriores mediante operadores.
- una asignación.
- una declaración.

Los cuatro primeros puntos son comunes en casi todos los lenguajes de programación. Que una asignación también sea una expresión es menos corriente, y se da en C. El último punto es especial de C++ y permite declarar variables en sitios insospechados, como en las «condiciones» de instrucciones condicionales o iterativas.

Ejemplo:

Constantes literales:

```
4   'a'   "Hola"   5.3e-3   0
```

Identificadores de objetos:

```
int a;
const double valor = 4.0;

a   valor
```

Llamadas a funciones:

```
getchar()   printf("Hola")
```

Combinación con operadores:

```
2 + 2      (i + j) / 2      'a' + 1      2.0 * sin(a)
```

Asignaciones:

```
c = getchar()      a = b = c = 0      j = 2 * fact(x)
```

Declaraciones:

```
if (int a = fu()) { /* ... */ }
for (int i = 0; i < n; i++) // ...
```

El resultado de una expresión se puede emplear como parte de otra (se habla entonces de una *subexpresión*.

El resultado de una expresión puede desecharse, escribiendo simplemente la expresión como una instrucción. Esto puede hacerse más explícito modelando la expresión al tipo vacío, *void*, pero no es necesario.

Las expresiones tienen un tipo de datos. En los casos donde en la expresión intervengan varios tipos de datos, el compilador convertirá unos a otros según las reglas de conversión.

Una llamada a una función que no devuelve nada no puede emplearse como subexpresión.

Ejercicios

E2.1. Para cada variable o función del código a continuación, diga su tipo de almacenamiento y su enlace; indique su ámbito y visibilidad.

```
int a;
const double pi = 3.14;
extern const int x;
static void f();
extern void g(int);
static bool b;
void h(register int x, int y)
{
  register int a;
  float f;
  if (int i == a)
    return;
  // ...
}
```

E2.2. Escriba un programa que muestre en la salida estándar los tamaños en bytes de los tipos fundamentales, unos pocos tipos puntero y unas pocas enumeraciones a su elección.

PISTA: Emplee el operador `sizeof`.

E2.3. Escriba un programa que muestre en la salida estándar las letras *a..z*, los dígitos *0..9* y sus valores enteros. Haga lo mismo para otros caracteres imprimibles. Repítalo usando notación hexadecimal. Para simplificar, suponga el código de caracteres ASCII (ISO–646 US), de 7 bits, donde las letras van ordenadas y consecutivas, así como los dígitos.

PISTA: Para la notación hexadecimal *inserte* en la salida el *manipulador hex*. Para saber si un carácter es imprimible puede emplear la función de la biblioteca estándar de C *isprint()*, declarada en `<cctype>`.

E2.4. Escriba una función que reciba dos *string*s y devuelva un *string* que sea la concatenación de los dos con un subrayado enmedio. Por ejemplo, si la función recibe `basic` y `string`, debe devolver `basic_string`. Vuelva a escribir la función pero sin emplear nada nuevo de C++, como si lo hiciera en C (*malloc()*, *strlen()*, etc.). Compare las dos funciones. ¿Cuál es más fácil de hacer, cuál es más rápida al ejecutarse y cuánto, cuál es más fácil de utilizar? ¿Merece la pena emplear cadenas de bajo nivel, o de alto?

PISTA: Para medir los tiempos de ejecución de las funciones emplee la función de la biblioteca estándar de C/C++ *clock()*. Como se ejecutarán rapidísimamente, mida con *clock()* no una sola ejecución, sino muchas iguales, en un bucle grande.

E2.5. Escriba una función *itos()* que reciba un *int* y devuelva su representación como un *string*.

E2.6. Cree un *vector<char>* que contenga las letras del alfabeto en orden. Imprima los elementos de ese *vector* en orden y en orden inverso.

E2.7. Cree un *vector<string>* y lea de la entrada estándar una lista de frutas; rellénelo con esta lista. Ordénela e imprímala en la salida estándar, escribiendo cada fruta en una línea precedida por un número de orden, empezando por 1.

Por ejemplo, pruebe con estas frutas:

```
naranja limón ciruela fresa manzana pera cereza melón
sandía melocotón caqui kiwi maracuyá plátano uva higo
```

PISTA: Para ir rellenando el vector, lo más fácil es tratarlo como una pila, e ir añadiendo elementos al final conforme se leen.

Para ordenar el vector puede emplear la función de la biblioteca estándar de C++ *sort()*, declarada en `<algorithm>`, pasándole como parámetros el principio y el final del vector. Éstos se obtienen mediante las funciones miembro *begin()* y *end()*, que son dos *iteradores* que se refieren evidentemente al principio y al final del *vector*. Esto se verá más en detalle en el capítulo sobre la biblioteca estándar.

Para no tener que escribir todas las frutas cada vez que pruebe el programa, escríbalas una sola vez en un fichero, por ejemplo `frutas.txt`, y cuando ejecute el programa redirija la entrada estándar a ese fichero. Así, si el ejecutable se llama `frutas`:

```
% frutas < frutas.txt
```

E2.8. Escriba en una instrucción o sentencia una expresión que borre el bit a 1 de más a la derecha de un entero sin signo x. Por ejemplo, *1011101110010000* se debe convertir en *1011101110000000*.

E2.9. En el siguiente código, ¿dónde da el primer error el compilador? ¿Hay más errores?

```
const
int a;
a = 100;
a++;
```

E2.10. Añada más operadores al programa de números complejos. Por ejemplo, multiplicación (`*`), división (`/`), asignaciones `+=`, `-=`, `*=` y `/=`, y los unarios `-` y `+`. Añada luego también la función que devuelva el módulo de un complejo.

Separe el programa en módulos: uno de cabecera con la definición de la estructura y las declaraciones de las funciones; otro `.cpp` con las definiciones de las funciones y operadores. Por último, otro con un pequeño programa de prueba. Haga un *makefile* para compilar.

Capítulo 3

Instrucciones de control

3.1. Introducción

Por lo general, un programa consiste en una secuencia de instrucciones que se ejecutan en el orden en que aparecen escritas. Este modelo de cálculo se conoce como «de ejecución secuencial».

Varias de las instrucciones de C++, que también están presentes en C aunque con algunas diferencias, permiten especificar que la siguiente instrucción a ejecutar sea otra distinta de la que le sigue en la secuencia. Esto se conoce como *transferencia de control*.

Este capítulo lo dedicaremos a analizar casi todas esas instrucciones que gobiernan el orden de ejecución y que son conocidas como de *control de flujo*. Las *excepciones*, que son un mecanismo avanzado de gestión de situaciones excepcionales, se estudiarán en un capítulo posterior.

3.2. Instrucciones y bloques

Lo primero que hay que saber es, aunque parezca evidente, que las instrucciones se ejecutan de izquierda a derecha y de arriba abajo.

Cuando se llama a una función, el control pasa a ésta hasta que termina o se llega a una instrucción `return`. Así, la llamada a función puede considerarse como la primera instrucción de control.

Las instrucciones pueden ser de dos tipos: simples o compuestas. Las instrucciones simples, como ya se habrá podido observar, acaban siempre en punto y coma; las compuestas son también llamadas *bloques* y constan de un conjunto de instrucciones (ya sean simples o compuestas) entre llaves. Su

sintaxis sería

```
{
  instrucción_1
  instrucción_2
  ...
  instrucción_n
}
```

y se puede utilizar en cualquier sitio en el que se pueda una instrucción simple, ya que un bloque es una instrucción.

Durante los años 60, se hizo patente que el empleo indiscriminado de transferencias de control era la principal causa de la complejidad encontrada en el desarrollo y mantenimiento de los programas. Todo apuntaba a la instrucción de salto incondicional (*go to*), que permite especificar una transferencia de control prácticamente a cualquier otro punto del programa. La que hoy se conoce como *programación estructurada* era entonces prácticamente sinónimo de *eliminación del salto incondicional*.

Jacopini y Bohm demostraron[1] que los programas se podían escribir sin utilizar ninguna instrucción de salto incondicional y empleando sólo tres estructuras de control: *composición secuencial*, *selección* y *repetición*.

C++, como la mayoría de los lenguajes modernos, proporciona varias instrucciones para implementar estas tres estructuras de control. La *estructura de composición secuencial* está construida internamente en el lenguaje, de forma que se ejecutarán automáticamente las instrucciones, una después de otra, en el orden en que aparezcan escritas. Para la estructura de selección proporciona las *instrucciones condicionales*; y, por último, para la estructura de repetición, las *instrucciones de iteración*.

3.3. Instrucciones condicionales

Cuando se desea ejecutar una instrucción dependiendo del valor de una expresión hay que utilizar una instrucción condicional. C++, al igual que C, proporciona tres instrucciones condicionales:

- Instrucción condicional simple
- Instrucción condicional compuesta
- Instrucción de decisión múltiple

[1] Teorema de Jacopini-Bohm.

3.3.1. Instrucción condicional simple

La sintaxis de una instrucción condicional simple es la siguiente:

```
if (expresión-lógica)
  instrucción
```

Esta instrucción se diferencia de su homónima en C en que la expresión de control es *lógica* y no *entera*. Es decir, que el resultado de la expresión es de tipo *bool* en lugar de *int*.

Así que cuando la expresión lógica se evalúe a verdadero (`true`), la instrucción se ejecutará, pero si la expresión fuera falsa (`false`), no.

Ejemplo:

```
if (precio < 10) cout << "¡Qué barato!" << endl;
if (precio > 30) cout << "¡Qué caro!" << endl;
```

Observe que la expresión debe ir siempre entre paréntesis y que, a diferencia de otros lenguajes, no existe la palabra reservada `then`.

Aunque la expresión que la instrucción `if` espera es lógica, puede también utilizarse una expresión numérica (real, entera, etc.) o de puntero. Esto es posible debido a que son transformadas en expresiones lógicas antes de evaluarse, con lo que el resultado de la expresión es un valor lógico. En general, puede emplearse cualquier cosa que se convierta implícitamente a *bool*.

Si *expr* es una expresión numérica o de puntero, lo siguiente

```
if (exp)
  instrucción
```

significa

```
if (exp != 0)
  instrucción
```

Es importante recordar que los operadores `&&` y `||` no evaluarán su segundo operando a menos que sea necesario[2] hacerlo. Esto es, si el primer operando es falso entonces el operador `&&` no necesita evaluar el segundo para saber que el resultado de la expresión es falso y por tanto no lo hace. Del mismo modo, si el primer operando del operador `||` se evalúa a verdadero, entonces no se evalúa el segundo, y el resultado de la expresión es verdadero.

[2]Es decir, la evaluación es condicional o «en cortocircuito».

Ejemplo:

```
void ordenar(int *v, int n)
{
  for (int i = 1; i < n; ++i) {
    int e = v[i];
    int j = i - 1;

    while (j >= 0 && v[j] > e) {
      v[j + 1] = v[j];
      --j;
    }
    v[j + 1] = e;
  }
}
```

La expresión `v[j] > e` sólo se evalúa si j es mayor o igual que 0. Por tanto, cuando j vale `-1` nunca llega a evaluarse el segundo operando, `v[-1] > e`, con lo que se evita acceder a una posición indebida del vector, ya que el elemento con índice `-1` no existe.

3.3.2. Instrucción condicional compuesta

La instrucción condicional compuesta se obtiene añadiendo la cláusula `else` (*en caso contrario*):

`if (`*expresión-lógica*`)`
  instrucción$_1$
`else`
  instrucción$_2$

En este caso la segunda instrucción se ejecuta si la expresión es falsa.

Ejemplo:

```
if (precio < 10)
  cout << "Barato" << endl;
else
  if (precio > 30)
    cout << "Caro" << endl;
  else
    cout << "Aceptable" << endl;
```

Como ya hemos dicho, las instrucciones pueden ser simples o compuestas; en el primer caso acabarán en punto y coma; en el segundo, las pondremos en un bloque, encerradas entre llaves.

Las instrucciones también pueden ser condicionales; entonces tenemos lo que se conoce como *cascada else-if*. Observe el sangrado que se suele utilizar para no formar una «escalera» que nos llevaría al límite de la pantalla o papel en el listado.

Ejemplo:

```
if (edad < 12)
  cout << "infancia";
else if (edad < 18)
  cout << "adolescencia";
else if (edad < 30)
  cout << "juventud";
else if (edad < 60)
  cout << "madurez";
else if (edad < 80)
  cout << "vejez";
else
  cout << "senectud";
```

Un `else` se asocia siempre al `if` más cercano. Con esto se evitan los problemas de ambigüedad sintáctica que se producen en el caso de las estructuras *else-if* múltiples.

Algunas instrucciones condicionales pueden sustituirse convenientemente por *expresiones condicionales*.

Ejemplo:

```
if (a <= b)
  max = b;
else
  max = a;
```

está mejor expresado del siguiente modo

```
max = (a <= b) ? b : a;
```

En este caso no es necesario encerrar la expresión entre paréntesis, aunque de esta forma es más fácil de leer.

3.3.3. Instrucción de decisión múltiple

La instrucción `switch` (*conmutar*) permite decisiones múltiples: se comprueba si una expresión entera concuerda con alguna *constante entera* de entre un conjunto de ellas, y se ejecutan instrucciones de acuerdo con dicha concordancia. La sintaxis básica es la siguiente:

```
switch (expresión-entera) {
  case expresión-constante-entera:
     instrucciones
  ...
  default:
     instrucciones
}
```

Las acciones que se llevan a cabo son las siguientes:

1. Se evalúa la expresión entera.
2. Se comprueba si concuerda con alguna de las expresiones constantes enteras que van tras la palabra reservada `case`.
3. Si hay concordancia, se salta a ese punto y se ejecutan las instrucciones correspondientes. Éstas pueden ser simples o compuestas; puede haber varias simples, acabadas en punto y coma como siempre. Puede incluso no haber ninguna.

   Tras ejecutarse las instrucciones del `case` concordante, *se siguen ejecutando las instrucciones que haya más abajo*, aunque correspondan a otro `case` o al `default`. Para evitar esto se utilizará la instrucción `break`, que nos sacará fuera del `switch`.
4. Si no hay concordancia, se ejecutan las instrucciones correspondientes a la *etiqueta* `default`, y se sigue como en el caso anterior, a menos que hayamos puesto `break`. La etiqueta `default` no tiene por qué ir obligatoriamente al final, pero de todas formas es costumbre ponerla allí.

   La etiqueta `default` es opcional; si no se pone y no hay concordancia, nos salimos del `switch` sin problemas, no se produce ningún error. No obstante, es buena práctica emplear siempre una etiqueta `default`.

Ejemplo:

```
enum { VERDE, AMBAR, ROJO } semaforo;
```

```
switch (semaforo) {
  case VERDE:
    cout << "Siga" << endl;
    break;
  case AMBAR:
    cout << "Precaución" << endl;
    break;
  default:
    cout << "Semáforo averiado" << endl;
    dar_alarma();
  case ROJO:
    cout << "Pare" << endl;
    break;
}
```

Observe que el `default` no lleva `break`: tras ejecutar sus instrucciones el flujo *cae*[3] al siguiente caso, que es el del semáforo en `ROJO`.

El último `break` sobra, pero es buena costumbre ponerlo por si añadimos algo tras él más adelante.

Alternativamente, una instrucción de decisión múltiple puede ser escrita como un conjunto de instrucciones condicionales.

Ejemplo:

```
switch (valor) {
case 1:
  f1(); break;
case 2:
  f2(); break;
default:
  f3(); break;
}
```

se podría expresar de la siguiente forma

```
if (valor == 1)
  f1();
else if (valor == 2)
  f2();
else
  f3();
```

[3]Esto se conoce como *fallthrough*.

El significado es el mismo, pero la versión con `switch` es preferible ya que es más apropiada, está especialmente pensada para comparar un valor con un conjunto de constantes.

3.3.4. Definiciones en las condiciones

Para evitar la utilización indebida de una variable, es una buena práctica introducir la variable en el ámbito más pequeño posible. Esto es, suele ser mejor retrasar la definición de una variable hasta el momento en el que haya que darle un valor inicial. De esta forma se evita utilizar la variable inadvertidamente antes de que se le asigne un valor.

Esto hace que, en ocasiones, sea conveniente definir variables en la expresión de una instrucción condicional.

Ejemplo:

```
if (double d = f())
  cout << d; // sólo se imprime si d != 0
```

La variable d ha sido definida e inicializada y el valor de d tras la inicialización es comprobado como valor de la condición.

El ámbito de d se extiende desde el punto en el que se define hasta el final de la instrucción controlada por la condición. Si hubiera una rama `else`, d estaría en el ámbito de las dos ramas.

La alternativa obvia y tradicional es definir d antes de la instrucción condicional. Sin embargo, esto abre el ámbito para el empleo de d antes de su inicialización o después de su vida útil prevista pudiéndose emplear accidentalmente para alguna otra actividad indebida.

```
double d;

t = d;  // empleo de 'd' antes de su inicialización
if (double d = f())
  cout << d;
d = 1;  // otro empleo de 'd'
```

Además de los beneficios lógicos de declarar las variables en las condiciones, al hacerlo se consigue también código fuente más compacto.

No obstante existe la restricción de que en las condiciones sólo se puede definir una variable o constante:

```
if (int i = 0, k = 5)        // ERROR
  ...
if (int i = 0; int k = 5)    // ERROR
  ...
```

La definición en las condiciones es una nueva característica incorporada en C++ que no está presente en C; esto es, en C no está permitido definir variables en las condiciones, causaría un error de compilación.

3.4. Instrucciones de iteración

La mayoría de los programas incluyen repeticiones o *iteraciones*. Una iteración es un grupo de instrucciones que se ejecutan de forma repetida, en tanto se satisfaga alguna condición de continuación de la iteración.

Dicho de otra forma, las instrucciones iterativas o *bucles* permiten repetir una serie de acciones hasta o mientras que se satisfaga cierta condición. Aunque con una bastaría, por comodidad tenemos varias, a elegir.

C++, al igual que C, proporciona tres tipos de estructuras de repetición:

- Instrucción `for` (*para*)
- Instrucción `while` (*mientras*)
- Instrucción `do-while` (*haz-mientras*)

3.4.1. Instrucción `for` (*para*)

La sintaxis de la instrucción `for` es la siguiente:

`for` (*expresión*$_1$`;` *expresión-lógica*$_2$`;` *expresión*$_3$`)`
  instrucción

Como de costumbre la instrucción puede ser simple o compuesta (bloque). Entre paréntesis aparecen tres expresiones separadas por dos signos de punto y coma. Para simplificar, puede pensarse que la primera es la de asignación de un valor inicial a una variable contadora, la segunda es la de control de dicha variable donde comprobamos su valor, y la tercera la de su incremento o decremento, quizá en una cantidad distinta de uno.

Esto hace que esté especialmente pensada para bucles regulares. La variable contadora, la condición de terminación y la expresión que actualiza la variable contadora se muestran en una sola línea[4]. Esto puede aumentar

[4]Si cabe, claro.

considerablemente la legibilidad del código y, por tanto, reducir errores.

Ejemplo:

Se va a escribir una función que calcule la n-ésima potencia de a. La función devolverá un entero que será a^n, y recibirá dos parámetros que serán a y n.

```
int potencia(int a, unsigned int n)
{
  int r = 1;

  for (int i = 0; i < n; ++i)
    r *= a;
  return r;
}
```

Otra forma sería:

```
int potencia(int a, unsigned int n)
{
  int r;

  for (r = 1; n != 0; --n)
    r *= a;
  return r;
}
```

El `for` de C++, al igual que el de C y a diferencia del resto de lenguajes como Pascal o FORTRAN, tiene dos particularidades:

1. Las expresiones pueden ser *cualesquiera*. No tienen por qué ser «asignación de un valor inicial», «control» e «incremento o decremento». Pueden involucrar llamadas a funciones, o cualquier operación.

   En ese caso a menudo la instrucción de iteración se expresa mejor con una instrucción `while`.

2. Cualquiera de las tres expresiones, o dos de ellas, o las tres, pueden omitirse.

   Si no es necesaria la *expresión inicial*, se puede omitir. Si se omite la *condición*, el bucle iterará indefinidamente a menos que se interrumpa con alguna instrucción de ruptura de control o de alguna otra forma como con un `return`, una llamada a *exit()* o el lanzamiento de alguna excepción.

   ```
   for (;;)    // bucle infinito
     ...
   ```

Definiciones en la instrucción `for`

En C++, una variable puede definirse en la primera expresión de una instrucción `for`. En ese caso el ámbito de la variable definida será hasta el final de la instrucción `for`.

```
for (int i = 0; i < n; ++i)
  ...
```

Si es necesario conocer el valor final de la variable contadora después de salir del bucle, la variable debe definirse fuera del bucle `for`.

No sólo es posible definir variables en la primera expresión, sino también en la segunda. Además se pueden definir varias variables separadas por comas en la primera expresión. Sin embargo está prohibido definir variables en la tercera expresión.

```
for (int i = 0; int j = 0; )                    // Bien
  ...
for (int i = 0, k = 0; int j = 0; )             // Bien
  ...
for (int i = 0; int j = 0; int k = 0)           // ERROR
  ...
```

3.4.2. Instrucción `while` (*mientras*)

Consta de la palabra reservada `while` seguida de la expresión de control entre paréntesis obligatorios, y una instrucción, que puede ser compuesta por supuesto. Ésta se repetirá hasta que la expresión de control sea falsa.

```
while (expresión-lógica)
  instrucción
```

La expresión de control se evalúa antes de ejecutarse la instrucción, por lo que si al entrar ya es falsa, aquélla no se dará nunca.

Cualquier instrucción escrita utilizando el bucle `for`

```
for (expresión1; expresión-lógica2; expresión3)
  instrucción
```

se puede reescribir utilizando el bucle `while` de la siguiente forma:

```
expresión1;
while (expresión-lógica2) {
```

```
  instrucción
  expresión3 ;
}
```

Y al revés, cualquier instrucción escrita con el bucle `while` se puede reescribir utilizando el bucle `for`. En general, se prefiere utilizar el bucle `while` al `for` cuando no hay una variable contadora en el bucle o cuando la actualización de la variable contadora se realiza en el cuerpo del bucle[5].

EJEMPLO:

Se mostrará la función anterior que calcula la n-ésima potencia de a, ahora utilizando una instrucción `while`.

```
int potencia(int a, unsigned int n)
{
  int r = 1;

  while (n != 0) {
    r *= a;
    --n;
  }
  return r;
}
```

Definiciones en la instrucción `while`

Las definiciones de variables en las instrucciones `while` se comportan de la misma forma que en las instrucciones condicionales. Se puede definir e inicializar una sola variable o constante.

3.4.3. Instrucción `do-while` (*haz-mientras*)

Análoga a `while` salvo que ahora la comprobación se hace al final, por lo que se garantiza que siempre se ejecuta la instrucción al menos una vez. Es parecido al *repetir-hasta* (*repeat-until*) de PASCAL, salvo que la comprobación se hace al revés (*mientras* en lugar de *hasta*).

```
do
  instrucción
while (expresión-lógica)
```

[5]Salvo si utilizamos `continue`, que en el caso del `for` salta a $expresión_3$.

Los significados de la expresión lógica y de la instrucción son los de costumbre. Para que en el listado uno no se confunda al ver el `while` y sepa que se corresponde con un `do` anterior, siempre se suele poner la instrucción en forma de bloque, aunque sea simple.

Ejemplo:

```
do {
    cout << "x = ";
    double x;
    if (cin >> x)
      cout << "x * x = " << x * x << endl;
} while (cin);
```

Las definiciones de variables en la condición de las instrucciones `do-while` están prohibidas, a diferencia de lo que ocurre con otras instrucciones de control.

3.5. Instrucciones de ruptura de control

A veces es conveniente tener la posibilidad de abandonar o salir de una instrucción de iteración durante su ejecución. Para ello contamos con tres instrucciones de ruptura de control:

- La instrucción `break` (*ruptura*)
- La instrucción `continue` (*continuar*)
- La instrucción `goto` (*ir a*)

3.5.1. Instrucción `break` (*ruptura*)

La instrucción

```
break;
```

hace que el control del programa se transfiera al fin del bucle o `switch` más interno que lo encierra. Se puede emplear para salir de un bucle «infinito», en lugar de comprobar el número en la expresión del `while` o del `for`, que generalmente es algo más complicado.

Ejemplo:

```
while (true) { // bucle infinito: como for (;;)
  int n;

  cout << "n = ";
  cin >> n;
  if (!cin || n == 0)
    break;
  cout << "Un número muy bonito, " << n << endl;
}
```

3.5.2. Instrucción `continue` (*continuar*)

La instrucción

```
continue;
```

provoca la vuelta a la expresión de control del `while` o a la tercera expresión del `for`, saltándose así el resto del cuerpo. Se emplea mucho menos que `break`.

Ejemplo:

```
for (int i = 0; i < n; ++i) {
  if (i % 2)  // si es impar
    continue;
  ... // procesar sólo los pares
}
```

Se suele emplear cuando la parte del cuerpo del bucle que sigue es complicada, de modo que negar la condición y sangrar otro nivel podría anidar profundamente el programa.

3.5.3. Instrucción `goto` (*ir a*)

C++ tiene el famoso `goto` que no hace falta nunca; sin embargo, hay casos en los que no ponerlo complicaría las cosas bastante. Un caso típico donde está *justificado* su empleo es aquél donde necesitamos salir de varios bucles anidados; `break` sólo nos sacaría al inmediato superior. También puede ser

importante en los casos en los que es esencial una gran eficiencia, por ejemplo, en el bucle interno de alguna aplicación en tiempo real o en el núcleo de un sistema operativo. La sintaxis es:

`goto` *etiqueta*`;`

La *etiqueta* es un identificador que ponemos antes o después, pero en la misma función donde esté el `goto`, seguido del signo de dos puntos y al menos una instrucción; si no queremos ninguna, tendremos que escribir la instrucción vacía: un punto y coma solitario, o dos llaves `{}`.

Ejemplo:

```
...
for (...)
  for (...) {
    ...
    if (desastre)
      goto panico;
    ...
  }
...
panico:
... // salvar lo que se pueda y salir
```

3.6. Funciones

A continuación se verá en detalle cómo se declaran y definen las funciones en C++.

3.6.1. Declaración: prototipo

Antes de llamar a una función, el compilador de C++ debe conocerla. Si no se ha definido antes de llamarla, esto se consigue mediante la declaración. En C++ la declaración se llama también *prototipo*, y es obligatoria.

> En C no hacía falta declarar una función antes de llamarla: el compilador suponía que devolvía un *int* y no se hacía cargo de comprobar los parámetros. También se podía declarar una función especificando sólo el tipo de devolución y no la lista de parámetros, con lo que el

> compilador no podía saber si la llamada era correcta. Esto provocaba errores difíciles de detectar.

Un prototipo de función informa al compilador del tipo devuelto por ella, de su nombre y de cuántos parámetros recibe y en qué orden, y de qué tipo es cada uno. Esto se consigue escribiendo el tipo de devolución, el nombre y la lista de parámetros formales, donde los nombres de éstos son opcionales y se consideran como comentarios.

En la llamada, el compilador comprobará los tipos de los parámetros reales y podrá dar mensajes de error si no coinciden o no pueden convertirse a los de los formales.

Ejemplo:

```
void fu1();            // no devuelve ni recibe nada
int fu2(void);         // devuelve un int, no recibe nada
fu3(int, float, char); // ERROR, falta el tipo de resultado
double fu4(int i, j);  // ERROR, falta el tipo de j
char* fu5(char* origen, char* destino);
```

Si la función no ha de recibir parámetro alguno, la lista puede estar vacía o constar solamente de la palabra `void`; la costumbre en C++ es no poner nada.

> Pero en C, la lista vacía no indica que la función no recibe parámetros, sino que la función recibe un número indeterminado de ellos: el compilador no tiene información y no puede comprobar la llamada.

Hay que poner el tipo de cada uno de los parámetros, aunque esté repetido, como se ve en el incorrecto prototipo de *fu4()*.

En *fu5()* se ve cómo añadir identificadores con nombres descriptivos en la lista de parámetros hace más fácil saber qué significan, aunque en este punto el compilador no haga caso de ellos.

A veces podemos querer que el compilador no compruebe los parámetros, bien todos o bien a partir de uno determinado. Para ello se puede poner la elipsis o puntos suspensivos.

Esto también es necesario para funciones que reciben un número indeterminado de parámetros, como la famosa *printf()* de la biblioteca de C.

Ejemplo:

```
int printf(const char* formato, ...);
int fprintf(const char* formato ...);
int scanf(const char* ...);
int f(...);
int g(..., char c);     // ERROR
int h(char*, ..., int); // ERROR
```

La elipsis debe ir sola o la última en la lista de parámetros; por eso las declaraciones de *g()* y *h()* están mal.

El compilador no comprobará los parámetros de *f()* en absoluto. Esto es equivalente a la lista vacía en C.

De las tres primeras funciones sólo se conoce el tipo del primer parámetro. Observe que se puede omitir la coma en C++[6].

3.6.2. Definición

En la definición de la función escribimos el código, las instrucciones que tiene que llevar a cabo. Este código va como un bloque o instrucción compuesta (el *cuerpo* de la función) tras la *cabecera*, que es como el prototipo, salvo que sí hay que nombrar los parámetros puesto que se van a utilizar en el cuerpo, y son variables locales `auto` (automáticas).

> Sin embargo, si no vamos a utilizar algún parámetro, podemos omitir su identificador. Esto puede ser útil en la fase de construcción del programa, cuando aún la función está en su fase inicial sin perfilar, o para dejar abierta una posibilidad de expansión[7]. Esta posibilidad de dejar un parámetro sin identificador evitará que el compilador advierta de que una variable no se ha utilizado. Esto no puede hacerse en C.

Dentro de la función, en el cuerpo, se devuelve un valor mediante la instrucción `return`, a la que sigue una expresión del tipo adecuado (o convertible); si la función no devuelve nada, `return` tendrá que ir solo, sin expresión.

> Aunque sí puede ponerse una «expresión de tipo *void*».

[6] Pero no en C; es ésta una diferencia que no comprendemos y vemos absurda.

[7] También es muy útil en la sobrecarga de los operadores post-incremento y post-decremento, como se verá.

EJEMPLO:

```
void g(int);

void f(int n)
{
  // ...
  return g(n);
}
```

Puede pensarse que esto es un tanto absurdo; podría haberse puesto el equivalente:

```
g(n);
return;
```

Pero esto es necesario para funciones genéricas (*plantillas*), donde el tipo de retorno sea paramétrico o genérico; es decir, un tipo cualquiera T, que unas veces podrá ser *void* y otras no.

Si se llega a la llave de cierre del bloque, el compilador «añade» automáticamente la instrucción `return`. Si la función tiene que devolver un valor y se llega al final de su bloque, se devuelve *basura* y el compilador advertirá de ello.

Una excepción es la función *main*, que según el Estándar debe devolver un *int*; si se llega a la llave de cierre de su cuerpo, el compilador añade

```
return 0;
```

Esta excepción no ocurre en C.

EJEMPLO:

```
int incremento(int a)
{
  return ++a;
}

double area_circulo(double radio)
{
  extern double pi;
  return pi * radio * radio;
}

void fu1()
{
```

```
  return 0;  // ERROR
}

int fu2()
{
  return;    // Mal
}

int fu4(void) // devuelve int, no recibe nada
{
  cout << "fu4()" << endl;
}        // Mal. Devuelve «basura» en vez de un int
```

Es posible definir funciones con un número variable de parámetros. Para ello hay que hacer uso de unas macros definidas en la cabecera estándar `<cstdarg>`. Consulte por ejemplo [2] o [8].

Ejemplo:

media.cpp

```
1  /*
2    Calcula la media de una ristra de reales terminados
3    con un valor que indica el primer parámetro.
4    Compile con -DTEST para probar la función.
5  */
6
7  #include <cstdarg>
8  #include <iostream>
9  using namespace std;
10
11 double media(double fin ...)
12 {
13   va_list ap;
14   va_start(ap, fin);
15   double total = 0.0;
16   int num;
17   double arg;
18   for (num = 0; (arg = va_arg(ap, double)) != fin; num++) {
19     // va_arg obtiene el sgute. arg. y lo compara con «fin»
20     total += arg;
21 #ifdef TEST    /* PRUEBA */
22     cout << arg << endl;
23 #endif
24   }
25   va_end(ap);
26   return total / num;
```

```
27  }
28
29  #ifdef TEST   /* PRUEBA */
30  int main()
31  {
32    double m = media(0.0, 1.0, 2.0, 5.3, 4.0, 4.0, 2.0, 0.0);
33    cout << "La media de 1, 2, 5.3, 4, 4 y 2 es: " << m << endl;
34  }
35  #endif
```

3.6.3. Uso de una función

Una vez declarada y definida, sólo se pueden hacer dos cosas con una función:

1. Llamarla.
2. Tomar su dirección.

Para llamarla, simplemente en una expresión se pone su nombre seguido de la lista de parámetros reales, entre paréntesis y separados por comas; si no ha de recibir ningún parámetro, se pondrán los paréntesis solos.

Para tomar su dirección, se le aplicará como es previsible el operador *dirección* (`&`) a su nombre, sin lista de parámetros. Pero no hace falta el operador `&`: el solo nombre de la función sin lista de parámetros produce su dirección de comienzo.

¿Y qué se puede hacer con la dirección de una función? Pasarla a otra función o asignarla a un puntero, lo cual es útil para construir por ejemplo *tablas de funciones* (esto es, vectores de punteros a funciones, que pueden llamarse a través del nombre del vector y un índice).

Ejemplo:

```
void (*pf)(int); // puntero a función que recibe un int
                 // y no devuelve nada
void func(int);  // declaración de la función

pf = &func;      // pf contiene la dirección de func()
pf =  func;      // forma alternativa, más corta

func(4);         // llamada a la función directamente
pf(4);           // llamada a través del puntero con su dirección
(*pf)(4);        // como antes, forma más explícita y larga
```

3.6.4. La funcion *main()*

Para el compilador, no tiene nada de especial, *main* no es una palabra reservada del lenguaje. Sin embargo, ese nombre sí es especial para el enlazador y para el sistema de apoyo de C++ en tiempo de ejecución; la rutina de arranque se encargará de cargar la función llamada *main()* en primer lugar cuando se ejecute el programa. De modo que en un programa completo en C++ debe haber una función, y sólo una, llamada *main()*.

Como ninguna otra función del programa llamará a *main()*, sino que desde ésta se llamará a las otras, no es preciso declarar *main()* y casi nunca se hace; basta con definirla.

Según el Estándar, *main()* devolverá un valor entero de tipo *int* al entorno de ejecución; el valor 0 indica que el programa se ha ejecutado correctamente, y un valor distinto indica que el programa ha terminado ordenadamente pero con algún tipo de error. Para evitar tener que escribir siempre la instrucción `return 0`, el compilador la «añadirá» automáticamente antes de la llave de cierre.

La rutina de inicialización y arranque, cuando llama a *main()*, le pasa además dos parámetros. El primero es un número entero, de tipo *int*, que representa el número de parámetros pasados al programa incluyendo su propio nombre (por lo que este valor siempre es 1 al menos). El segundo es la dirección de un vector de punteros a cadenas de caracteres, conteniendo cada una un parámetro.

Dependiendo de si vamos a hacer uso de estos parámetros o no, la definición de *main()* será pues una de las siguientes:

```
int main() { /* ... */ }
int main(int argc)  { /* ... */ }
int main(int, char** argv)  { /* ... */ }
int main(int argc, char* argv[])  { /* ... */ }
```

En la primera forma, no se van a utilizar los parámetros. En la segunda, sólo se va a utilizar el primer parámetro. En la tercera, sólo se va a usar el segundo parámetro. Y en la cuarta, se van a utilizar los dos.

Los nombres *argc* (por *argument count*, «número de parámetros») y *argv* (por *argument values*, «valores de los parámetros») son arbitrarios, pero es una costumbre general que sean éstos. El tipo del segundo parámetro real, *argv*, es *char*`*` `[`*argc* `+ 1]`. Como el mecanismo de llamada a *main()* es el mismo que en el lenguaje C, se emplea un vector (de bajo nivel) de punteros a cadenas de caracteres acabadas en 0 (también de bajo nivel). Se cumple que *argv*`[0]` apunta al nombre del programa, *argv*`[`*i*`]` apunta al parámetro *i*-ésimo y *argv*`[`*argc*`]` es 0, el puntero nulo.

La definición de *argv* puede ser indistintamente una de las dos mostradas más arriba, con notación de vector o de puntero, puesto que una función no recibe realmente un vector de bajo nivel sino la dirección de su comienzo, y una dirección es de tipo puntero.

El tipo es obligatorio en C++:

```
int main() { /* ... */ }
int main(int argc, char* argv[]) { /* ... */ }
...
```

3.6.5. Sustitución «en línea»

En C, las funciones cortas y utilizadas frecuentemente se suelen sustituir por macros del preprocesador, pues al sustituirse el código en la «llamada» se gana en eficiencia: el compilador no tiene que generar código para la llamada a la función y no tiene que copiar parámetros ni el valor de devolución.

Sin embargo el compilador tampoco puede comprobar la corrección de los parámetros reales, y se pueden producir efectos colaterales peligrosos si el programador no es cuidadoso. Además no existe con las macros el concepto de ámbito, lo que hace inútil las definiciones dentro de espacios de nombres o de clases.

Para evitar estos inconvenientes, en C++[8] se introduce la palabra reservada `inline`, que actúa indicando o sugiriendo al compilador que expanda el código de la función «en la línea» de la llamada, de ahí el nombre. Aparte de esto, una función `inline` se comporta como cualquier otra: el compilador comprobará la corrección de los parámetros y del valor devuelto, efectuando en su caso las conversiones necesarias.

Para que una función sea `inline` hay que poner esta palabra en la definición de la función; *no tiene efecto en la declaración*. Además una función `inline` tiene enlace interno (§2.2.2); esto es, es `static` y sólo es visible en el fichero donde se incluya. Si se define en dos ficheros, las definiciones deben coincidir exactamente. Por esto, casi siempre se definirá una función `inline` en un fichero de cabecera. Éste y el de las plantillas es el único caso en el que se deben definir funciones en ficheros de cabecera.

Ejemplo:

```
inline int doble(int x) { return x + x; }
```

[8] Y en algunos compiladores de C, como una extensión.

Cuando el compilador ve la definición de una función `inline`, coloca el tipo de la función (esto es, su lista de parámetros, llamada también *signatura*, y su tipo de retorno) *más* el cuerpo de la función en su tabla de símbolos. Cuando se llama a la función, el compilador se asegura de que la llamada sea correcta y el valor devuelto se emplee adecuadamente, y entonces sustituye el cuerpo de la función por su llamada. El código de una función ocupa normalmente más espacio que su llamada, pero si es lo bastante pequeño puede que no.

La palabra `inline` no es una orden para el compilador sino una sugerencia, como `register` (§2.2.2): el compilador puede hacer caso o no, dependiendo de lo bueno que sea, de lo grande que sea la función, etc. En general, un compilador rehusará la sustitución en línea de una función si:

- es la función *main()*.
- es recursiva.
- contiene iteraciones o saltos.
- contiene instrucciones en ensamblador (`asm`).
- es muy grande, tiene muchas instrucciones.
- contiene destructores locales.
- es virtual.

Los dos últimos puntos se verán en su momento.

> Por otra parte algunos programadores dicen que no merece la pena definir `inline` las funciones que hacen entrada/salida, por muy pequeñas que sean aparentemente, puesto que el tiempo ahorrado con la sustitución en línea es despreciable frente al de las operaciones de entrada/salida.
>
> Otros dicen que estas optimizaciones son cosa del compilador, no del programador, por lo que optan por no emplear nunca `inline` y dejan que sea el compilador el que optimice (todos tienen opciones de optimización). Además se expone el código de una función `inline` a la vista de todos cuando se define en un fichero de cabecera.
>
> Sin embargo, al definir una *clase* de objetos, se verá que es muy común que haya bastantes funciones muy cortas, por lo que será apropiado definirlas `inline`. En general creemos que, fuera de las clases, las funciones `inline` deben simplemente sustituir o hacer el papel que harían las macros del preprocesador.

3.6.6. Parámetros predeterminados

A veces una función tiene un parámetro que muchas veces toma el mismo valor. Por ejemplo, podríamos tener una función que calculara el logaritmo de un número en cualquier base y la mayoría de las veces la llamáramos para calcular un logaritmo decimal. O una función que devolviera una cadena de caracteres con la representación de un número en cualquier base de numeración, y casi siempre la usáramos para pasar a la base binaria. U otra que elevara un número entero a una potencia, y lo más frecuente fuera elevar al cuadrado.

Para no tener que estar siempre repitiendo el mismo parámetro en la llamada, C++ (pero no C) permite que las funciones tengan parámetros predeterminados, o por omisión; simplemente se declaran en el prototipo con su valor prefijado. En la llamada, se pueden omitir esos parámetros, y automáticamente tomarán el valor predeterminado.

Ejemplo:

Los prototipos de las funciones antedichas serían

```
double logaritmo(double x, double base = 10.0);
string cambiabase(int num, int base = 2);
int potencia(int base, int exponente = 2);
```

Y las llamadas podrían ser:

```
double log2 = logaritmo(2.0); // igual que  logaritmo(2.0, 10.0)
string binario = cambiabase(65535); // como cambiabase(65535, 2)
int c = potencia(12);         // 12 al cuadrado: potencia(12, 2)
```

Cuando el compilador ve que falta el último parámetro y que todo lo demás está bien, sustituye automáticamente el parámetro que falta por el valor especificado por omisión en la declaración.

Hay dos reglas que hay que cumplir a la hora de declarar una función con parámetros por omisión:

1. Sólo los parámetros de más a la derecha pueden tomar valores predeterminados. Esto es, no se puede tener un parámetro prefijado seguido de otro sin prefijar. Otra forma de decirlo: una vez que se empiezan a poner parámetros prefijados, hay que seguir hasta el final.

EJEMPLO:

```
void f(int a, int b, int c = 2, int d = 4);         // Bien
void g(int a, int b = 1, int c, int d = 4);         // ERROR
void h(int a = 0);                                  // Bien
void k(int a = 0, int b = 1, int c = 2, int d = 4); // Bien
```

2. Los parámetros predeterminados deben especificarse en la declaración o prototipo solamente, no en la definición, a menos que sirva también de declaración; es decir, el compilador debe conocer el parámetro predeterminado antes de la llamada.

EJEMPLO:

```
void f(int a, int b = 0);   // declaración
...
f(3);      // Bien. Igual que f(3, 0); a = 3, b = 0
f(4, 1);   // Bien. a = 4, b = 1
...
void f(int a, int b) // definición: NO se repite «b = 0»
{
  ...
}
```

Algunos programadores, no obstante, tienen la costumbre de escribir de nuevo los valores predeterminados en la definición, entre comentarios al estilo de C, como documentación adicional:

```
void f(int a, int b /* = 0 */) { /* ...definición... */ }
```

Es posible, como de costumbre, no escribir los identificadores de parámetros en la declaración:

EJEMPLO:

```
void fu(int x, int = 0, float = 1.1f);
```

Y ya se ha visto que en C++ tampoco hacen falta en la definición, siempre naturalmente que no se utilicen:

Ejemplo:

```
void fu(int x, int, float f) { /* ... */ }
```

Dentro del cuerpo de la función *fu()*, x y f pueden emplearse, pero no el parámetro de enmedio, puesto que no tiene nombre. Las llamadas podrían ser:

```
f(1);           // Bien
f(1, 0);        // Bien, pero inútil. Igual a lo anterior.
f(1, 2, 3.0f);  // Bien
f(1, , 3.0f);   // ERROR
```

La idea de esto es que uno puede querer cambiar la definición de la función para emplearla más adelante, sin tener que cambiar las llamadas a la función. Por supuesto se puede hacer dando nombre a todos los parámetros, pero así se evita que el compilador avise de que hay una variable sin utilizar.

Más importante: si uno empieza a utilizar un parámetro de la función y más tarde decide que ya no le hace falta, puede quitarlo sin obtener avisos a la hora de compilar y sin que el código cliente que llame a la función tenga que modificarse.

3.6.7. Sobrecarga

Cuando uno crea un objeto, una variable, le da un nombre a una región de almacenamiento. Cuando crea una función, le da un nombre a una acción. Es muy importante elegir nombres adecuados, pues así el programa será más fácil de leer y entender. Surge un problema cuando se intenta asociar el concepto de matiz en un lenguaje humano con un lenguaje de programación. A menudo una palabra expresa varios significados, según el contexto. En lingüística esto se llama «polisemia» y en programación, «sobrecarga». La sobrecarga es muy útil sobre todo para expresar diferencias triviales. Uno dice: «lava el coche», «lávate la cara»; sería tonto tener que decir: «coche_lava el coche», «cara_lava la cara» sólo para que el oyente no tuviera que hacer distinción entre las acciones pedidas. Los lenguajes humanos son redundantes, de forma que si falta alguna palabra, aún se puede determinar el significado. No necesitamos identificadores únicos: podemos deducir el significado del contexto.

Pues algo como lo del ejemplo es lo que pasa en muchos lenguajes de programación, como en C. Así, si uno tiene tres tipos de datos diferentes que quiere mostrar, como *int*, *char* y *float*, hay que crear tres funciones diferentes; por ejemplo, *mostrar_int()*, *mostrar_char()* y *mostrar_float()*. Esto supone trabajo extra tanto para el programador como para el lector.

Modificación de nombres

Aunque la sobrecarga de funciones significa que podemos definir varias funciones con el mismo nombre, el compilador sigue con la regla de que los identificadores en el mismo ámbito han de ser únicos. Para evitar esta contradicción, el compilador asigna internamente nombres a las funciones diferentes de los dados por el programador, de forma que no haya dos identificadores iguales, al menos para el compilador. Esto es lo que se conoce como *modificación* o *decoración de nombres*[9]. No hay ninguna norma acerca de cómo un compilador ha de generar esos nombres; esto puede dar problemas a la hora de adquirir una biblioteca para un compilador y enlazador particular y pretender usarla con otro, pero esos problemas también se darían de todas formas debido a la manera en que compiladores diferentes generan código.

Ejemplo:

En el programa siguiente, la función *mostrar()* está sobrecargada[10]:

sobrecarga.cpp

```
1  // Ejemplo de sobrecarga
2
3  #include <iostream>
4  using namespace std;
5
6  void mostrar(char c)   { cout << "Carácter: " << c << endl; }
7  void mostrar(int i)    { cout << "Entero  : " << i << endl; }
8  void mostrar(double d) { cout << "Real    : " << d << endl; }
9
10 int main()
11 {
12   mostrar('A');
13   mostrar(65);
14   mostrar(65.);
15 }
```

En el sistema donde se ha escrito y probado[11], el compilador ha generado internamente los nombres

```
mostrar__Fc    mostrar__Fi    mostrar__Fd
```

[9] En inglés, *name mangling*.

[10] En versiones antiguas de C++ existía la palabra reservada `overload` (*sobrecargar*), pero pronto se eliminó por superflua.

[11] Linux 2.2.4, *gcc version egcs-2.91.60 Debian 2.1 (egcs-1.1.1 release)*.

lo cual puede averiguarse mediante el programa `nm` o mirando el código en ensamblador generado por la opción `-S` del compilador. De todas formas, esto es transparente al programador, que no tiene por qué preocuparse de cómo sean estos nombres.

Hay un beneficio más en la modificación de nombres, aparte de permitir la sobrecarga; se llama el *enlazado con seguridad de tipos*. Se refiere esto al problema que surge cuando uno declara erróneamente una función. En C, esto es muy común puesto que uno puede no declararla en absoluto, o hacerlo sin parámetros; en C++ esto es más difícil porque toda función ha de declararse con su prototipo antes de ser llamada, pero aun así se podría dar el caso de declararse mal, bien por hacerlo «a mano» y equivocarse, bien por emplear una cabecera desfasada. Observe lo siguiente.

Ejemplo:

def.cpp

```
1 // Definición de función
2 void f(int i) {}
```

uso.cpp

```
1 // Declaración errónea de función
2 void f(char);
3
4 int main()
5 {
6   f(1); // ERROR de enlazado en C++, pero no en C
7 }
```

Aunque se vea que la función es realmente `f(int)`, el compilador no lo sabe al compilar `uso.cpp` porque se le ha dicho erróneamente que es `f(char)`, así que la compilación está bien. En C, el enlazado tampoco daría error, pero sí en C++. Puesto que el compilador disfraza los nombres, la función es algo como `f__Fi`, mientras que el uso dado sería `f__Fc`. Cuando el enlazador intenta resolver la referencia a `f__Fc`, sólo puede encontrar `f__Fi`, que no es la misma, y da un error. Aunque este problema no es muy frecuente, cuando ocurre es dificilísimo de detectar, sobre todo en un programa grande. Éste es uno de los casos donde se puede encontrar un error difícil en C simplemente compilando con un compilador de C++.

Signatura

La signatura consiste en el nombre de una función más su lista de parámetros. Otra forma de decirlo: el prototipo menos su tipo de retorno. Para sobrecargar una función, la signatura ha de ser diferente. Esto es, las funciones sobrecargadas[12] deben diferir en su lista de parámetros, bien en número o en tipo o en ambas cosas.

Una pregunta típica en este punto es: «Bien, eso es evidente, pero ¿por qué no pueden diferenciarse solamente en el tipo de retorno?» Supongamos:

```
void f();
int f();
```

Esto podría funcionar bien si el compilador pudiera determinar inequívocamente el significado a partir del contexto, como en

```
int x = f();  // La 2ª: int f();
```

pero no. Porque en C++, así como en C, uno puede siempre desechar el valor de una expresión («conversión de línea», a *void*). Entonces, ¿cómo podría distinguir el compilador lo siguiente?

```
f();  // ¿«void f()» o «(void)int f()»?
```

Aún peor: no ya el compilador, sino ni el lector del código sabría a qué función se está refiriendo la llamada. La sobrecarga basada en el tipo de retorno no sería imposible de implementar, pero llevaría a problemas que no merece la pena sufrir, así que se decidió no permitirlo, de forma que la sobrecarga sea independiente del contexto.

¿Sobrecarga o parámetros prefijados?

Ambas cosas son una comodidad para el programador[13], y en realidad los parámetros predeterminados son otra forma de sobrecarga: en ambos casos se nos permite emplear un mismo nombre en situaciones diferentes.

¿Cuándo emplear una u otra característica? En general los valores predeterminados son útiles para que el compilador los sustituya cuando nosotros

[12] Siempre nos referimos a funciones en el mismo ámbito. Está claro que dos funciones pueden perfectamente llamarse igual si están definidas con enlace interno en ficheros diferentes (`static`) o están en espacios de nombres distintos, o en clases distintas, como se verá.

[13] Aunque la sobrecarga no es sólo eso, sino una necesidad, como se verá cuando se hable de constructores.

no queramos dárselos por comodidad. Cuando dos funciones tienen comportamientos muy parecidos, lo más lógico suele ser emplear parámetros predeterminados, y al revés. Si se comprueba dentro de una función el valor predeterminado de un parámetro para hacer algo diferente, es que se debería haber sobrecargado la función.

Ejemplo:

La siguiente función es un **mal** ejemplo del empleo de parámetros predeterminados.

```
void f(int i = 0)  // declaración y definición
{
  if (i == 0) {
    // A: código para cuando i == 0
  }
  else {
    // B: código para cuando i != 0
  }
}
```

Debería haberse escrito mejor:

```
void f(void)
{
  // A
}

void f(int i)
{
  // B
}
```

Resolución de ambigüedades en la sobrecarga

Por lo visto hasta ahora, la sobrecarga no parece muy difícil. Y no lo es si uno anda con mucho cuidado a la hora de escoger qué función sobrecargar y cómo, y a la hora de llamarla. Porque si los parámetros de la llamada no coinciden exactamente, el compilador intentará convertir los tipos, y entonces pueden darse ambigüedades; o sea, errores. A la hora de buscar a qué función sobrecargada llamar, el compilador de C++ intenta lo siguiente por orden:

1. Coincidencia exacta o trivial.

2. Promociones.
3. Conversiones estándar.
4. Conversiones definidas por el usuario.
5. Coincidencia con elipsis (...).

Siempre hay que procurar ser muy cuidadoso y evitar en lo posible las conversiones implícitas.

Uso de funciones no escritas en C++

Muchas veces un programa en C++ contiene partes escritas en otros lenguajes, sobre todo en C: es muy común que desde C++ se llame a funciones de la biblioteca de C. Sin embargo cada lenguaje tiene sus propios convenios a la hora de poner los parámetros de las funciones en la pila, de usar los registros, etc. Por ejemplo, en C, como no hay sobrecarga, tampoco existe la modificación de nombres recién vista (§3.6.7). Para ayudar en este enlace entre lenguajes, a la hora de declarar una variable o (sobre todo) función externa escrita en otro lenguaje, se puede especificar un convenio de enlazado como una cadena de caracteres especial entre la palabra `extern` y la declaración. Así por ejemplo, la función de la biblioteca estándar de C (y C++) *exit()* se declararía:

```
extern "C" void exit(int);
```

Los únicos convenios de enlazado reconocidos en el Estándar son `"C"` y por supuesto `"C++"`, que no tiene efecto; este *C* no se refiere al lenguaje C —la función podría haber estado escrita en FORTRAN o ensamblador— sino a un convenio de enlazado que siga las normas del enlazado en C; por ejemplo, no habrá modificación de nombres (y por lo tanto no se podrá sobrecargar).

Si hay que declarar muchas funciones así, como ocurre en un fichero de cabecera, se puede utiliar un *bloque de enlazado*:

```
extern "C" {
  void exit(int);
  int atexit(void (*)(void));
  // ...
}
```

Esta técnica se puede utilizar para producir un fichero de cabecera válido para C++ a partir de uno de C:

```
extern "C" {
  #include <stdlib.h>
}
```

Y, combinado con la compilación condicional, se suele emplear para producir ficheros de cabecera válidos tanto para C como para C++:

```
#ifdef __cplusplus
extern "C" {
#endif
  void exit(int);
  int atexit(void (*)(void));
  /* ... */
#ifdef __cplusplus
}
#endif
```

3.6.8. Recursividad

La recursividad es un mecanismo de definición presente en la mayoría de los lenguajes de programación modernos que permite, bajo ciertas condiciones, que las funciones se definan en función de sí mismas.

En particular, esto implica que se permite a una función realizar una o más llamadas a sí misma: estas llamadas se denominan *recursivas*.

Para que una función recursiva esté bien definida, debe existir una *medida*[14] de los parámetros de la función que decrezca estrictamente en cada llamada recursiva respecto a una *relación de orden bien fundada*[15]. Un orden está «bien fundado» si no existe una secuencia infinita de elementos estrictamente decreciente.

Esto garantiza que existe una función (en sentido matemático), y sólo una, que satisface la definición.

Si la definición está bien hecha, y los parámetros de entrada son correctos, la secuencia de llamadas recursivas termina en alguna llamada que no genera otras.

Estas *llamadas terminales*, al acabar su ejecución, devuelven el control a la llamada anterior que, a su vez, finalizará en algún momento devolviendo el control a la anterior, y así hasta que la secuencia, y la propia llamada inicial, acaben.

[14] También llamada *función limitadora* o *de cota*.
[15] También llamada *noetheriana*. En realidad basta con un preorden.

Ejemplo:

Un caso clásico de este tipo de diseño es el cálculo del factorial de un número. A continuación se presenta una función recursiva que lo calcula:

```
unsigned int factorial(unsigned int n)
{
  return n ? n * factorial(n - 1) : 1;
}
```

Como se ve en el ejemplo, tenemos que definir unos valores base en los que no se realiza ninguna llamada a la propia función que se esté definiendo. Y existe una medida de los parámetros, $m(n) = n$, que decrece estrictamente en cada llamada recursiva respecto al orden bien fundado, $<$, de los números naturales:

$$\forall n \in \mathbb{N}\ [n > 0 \Rightarrow m(n-1) < m(n)]$$

Hay que tener en cuenta que una función también puede ser recursiva, aun sin haber llamadas recursivas directas en su definición, si hace referencia indirectamente (a través de otras funciones) a sí misma. En tal caso se dice que es *indirectamente recursiva*.

Cuando una función genera en cada llamada a lo sumo una llamada recursiva, diremos que dicha función es *recursiva lineal* o *recursiva simple*. En caso contrario, diremos que es *recursiva no lineal* o *recursiva múltiple*.

También decimos que dos funciones *f()* y *g()* son *mutuamente recursivas* si ambas son indirectamente recursivas a través de la otra.

No obstante, hay que tener cuidado con estas definiciones ya que se trata de una distinción de carácter *dinámico*. Podría ocurrir que en el cuerpo de una función recursiva lineal aparecieran varias llamadas recursivas, cada una en una alternativa diferente de una instrucción condicional. La recursividad seguiría siendo lineal ya que, en tiempo de ejecución, las alternativas son mutuamente excluyentes y, a lo sumo, se produciría una llamada.

Ejemplo:

```
int potencia(int x, unsigned int n)
{
  if (n == 0)
    return 1;
  else if (n % 2 == 1)
     return potencia(a * a, n / 2) * a;
   else
```

```
    return potencia(a * a, n / 2);
}
```

En cambio, en el siguiente ejemplo la función sería recursiva múltiple:

EJEMPLO:

```
unsigned int f(unsigned int n)
{
  if (n <= 1)
    return n;
  else
    return 5 * f(n - 1) - 6 * f(n - 2);
}
```

Hay que tener en cuenta que las funciones recursivas no son necesariamente eficientes, ya que mantienen valores que quedan en suspenso mientras la recursión termina y que necesitan ser almacenados en una pila para su empleo posterior.

Una función se dice *recursiva final*[16] cuando a la llamada recursiva no sigue ninguna otra instrucción ni su valor es empleado en otra operación antes de su devolución.

EJEMPLO:

```
unsigned int g(unsigned int n, unsigned int p = 1)
{
  return n ? g(n - 1, p * n) : p;
}
```

La función *g()* es una generalización del factorial. La llamada `g(n)` da al parámetro *p* el valor inicial `1`, haciendo que se calcule exactamente el factorial de *n*. Esto es, las llamadas `g(n)` y `factorial(n)` del ejemplo anterior son equivalentes. La diferencia estriba en que *g()* es recursiva final.

La ventaja de las funciones recursivas finales radica en que son más eficientes que sus equivalentes no finales y en que su transformación a función iterativa es automática.

[16] También *recursiva por la cola* o *por la derecha*.

Desde luego, la recursividad tiene exactamente la misma potencia expresiva que la iteración, en el sentido de que se pueden describir las mismas funciones con ambas. Sin embargo, su empleo es más adecuado para cierto tipo de problemas. De hecho, existen lenguajes que prescinden casi por completo de la iteración y utilizan la recursividad como único mecanismo de control.

Ejercicios

E3.1. ¿Qué imprimiría el siguiente fragmento en los casos en que *precio* vale 5, 26 y 34?

```
if (precio > 10)
  if (precio < 30)
    cout << "Está en el rango normal.";
else
  cout << "¿O no?";
cout << endl;
```

¿Está bien sangrado el fragmento de código anterior?

E3.2. Compruebe qué es lo que imprime el siguiente fragmento de código si t vale 20.

```
if (22 < t < 25)
  cout << "Hace una temperatura agradable.";
else
  cout << "Ese tiempo es insoportable.";
cout << endl;
```

¿Es lo que esperaba?

E3.3. ¿Hay algún problema con el siguiente bucle? Si lo hay, proponga distintas formas de solucionarlo valorando sus pros y contras.

```
for (size_t i = n; i >= 0; --i)
  ...
```

E3.4. Escriba un programa que lea números de su entrada mientras ésta sea válida y que, después, muestre la media aritmética de dichos números.

E3.5. ¿Es correcto el siguiente fragmento? Intente adivinarlo; luego pruébelo, y si no es correcto, corríjalo.

```
void mostrar(int i)    { cout << "Entero: " << i << endl; }
void mostrar(double d) { cout << "Doble : " << d << endl; }
...
mostrar('A');
mostrar(2);
mostrar(2.2f);
mostrar(2.2)
```

E3.6. Genere todas las combinaciones de apuestas simples de la *lotería primitiva* (6 números elegidos de entre 49) e imprima su número total. Debe obtener un total de $\binom{49}{6} = 13983816$.

E3.7. Escriba las funciones recursivas clásicas para calcular:

a) $n!$

b) El n-ésimo número de Fibonacci

Haga un programa de prueba que muestre una tabla de valores. ¿Qué ocurre en cada caso, y por qué, con valores grandes de n?

E3.8. Deseamos calcular la potencia de un número natural. Escriba tres versiones:

a) Una función recursiva trivial (no final)

b) Una función recursiva final

c) Una función iterativa

Escriba también un programa que muestre una tabla con las primeras potencias de 10, comprobando que con las tres versiones se obtiene el mismo resultado.

Capítulo 4

Clases y objetos

4.1. Introducción

La idea fundamental que subyace tras el paradigma de la orientación a objetos es la de organizar el sistema que se está desarrollando en torno a los objetos que intervienen en él.

Estos objetos se abstraen en clases. Mientras que un objeto es una entidad que posee un conjunto de datos y de operaciones, una clase es una descripción general que permite representar un conjunto de objetos similares.

Por definición, todos los objetos que existen dentro de una clase comparten los mismos datos y operaciones. La clase encapsula las abstracciones de datos y operaciones necesarias para describir un cierto tipo de entidades del mundo real.

En la terminología de la programación orientada a objetos, a los datos de una clase se les denomina *atributos* y a sus operaciones, *métodos*. Como se verá, los atributos pueden ocultarse de manera que la única forma de operar sobre ellos sea a través de alguno de los métodos que proporciona la clase. Esto permite ocultar los detalles de implementación, facilita el mantenimiento y favorece la reutilización.

4.2. Estructura de una clase

En C++ las clases se introducen a partir de una extensión de las estructuras de C (`struct`) mediante la que se permite que éstas contengan no sólo *miembros de datos*, sino también *funciones y operadores miembro*. A las variables de estos tipos se las denomina *objetos* y se puede acceder a sus miembros tal

y como se hacía con una estructura de C, independientemente de que sean datos o funciones.

Cuando una función u operador miembro se aplica a un objeto de una clase, implícitamente trabaja sobre sus miembros de datos. Así, cuando en la definición de una función miembro se hace referencia a un miembro de datos sin más, éste corresponde al del objeto sobre el que en cada momento se aplique la función. Esto es así porque, como veremos, recibe implícitamente un parámetro adicional: la dirección del objeto al que se aplica.

Mientras que los miembros de datos son únicos para cada objeto, ya que determinan su estado interno, las funciones miembro de la clase son compartidas por todos sus objetos, puesto que representan comportamientos comunes a todos ellos.

La definición de una clase es análoga a la de una estructura, sustituyendo la palabra reservada `struct` por `class`. La sintaxis básica se muestra a continuación:

```
class nombre {
public:
  sección pública
private:
  sección privada
};
```

Como veremos, la posibilidad de declarar secciones públicas y privadas dentro de la clase es lo que permite hacer uso del principio de ocultación de datos.

Además de miembros de datos y funciones miembro es posible definir dentro de una clase tipos de datos y, específicamente, otras clases. Una clase definida dentro de otra se denomina *clase anidada*.

4.2.1. Control de acceso

En una clase, por omisión, los miembros están ocultos al exterior. En cambio, todos los miembros de una estructura son visibles desde el exterior para mantener así la compatibilidad con C. Ésta es, en realidad, la única diferencia que existe en C++ entre las palabras reservadas `class` y `struct`.

La visibilidad permite controlar el acceso a los miembros, ya que sólo se puede acceder a los miembros que son visibles desde el exterior, quedando prohibido el acceso a aquéllos que están ocultos.

Evidentemente, se hace necesario un mecanismo que permita especificar qué partes de una clase son accesibles desde el exterior y cuáles no. Para lograr esto existen tres formas de control de acceso:

Público Visible desde el exterior.

Privado Oculto al exterior.

Protegido Se explicará al hablar de la herencia.

Cada uno de estos modos de controlar el acceso se especifica, respectivamente, con las palabras reservadas `public`, `private` y `protected`. Pueden existir tantas secciones públicas, privadas y protegidas como se deseen (incluyendo la posibilidad de que no haya ninguna).

> Lo recomendable es que sólo exista una sección de cada tipo; sin embargo, al permitirse que exista más de una se facilita la construcción de herramientas de generación automática de código (como traductores o herramientas de desarrollo rápido de aplicaciones). Éstas pueden añadir secciones al final de la clase, de manera incremental, conforme va haciendo falta. Evidentemente, esto afecta a la legibilidad del código generado, pero en muchas aplicaciones éste es interno y no está pensado para su comprensión o modificación posterior.

Ejemplo:

A continuación se define la clase *Semaforo*, que consta de una parte pública, en la que se encuentra una operación que permite cambiar cíclicamente el estado del semáforo, y de una parte privada, en la que se define un tipo enumerado con tres constantes de enumeración y un dato que representa el color actual del semáforo.

```
class Semaforo {
public:
  // ...
  void cambiar();
private:
  enum Color { ROJO, VERDE, AMBAR };
  Color c;
};
```

Se podría haber definido, equivalentemente, de la siguiente forma:

```
class Semaforo {
  enum Color { ROJO, VERDE, AMBAR };
  Color c;
public:
  // ...
  void cambiar();
};
```

o incluso empleando una estructura:

```
struct Semaforo {
  // ...
  void cambiar();
private:
  enum Color { ROJO, VERDE, AMBAR };
  Color c;
};
```

No obstante, la primera forma suele ser la preferida en C++ para definir una clase, ya que lo primero que se ve al leer la definición son los servicios que ofrece la clase al exterior, quedando para el final sus detalles internos. Se emplea la palabra reservada `class`, primero aparecen los miembros públicos y, luego, los privados.

Una vez definida la clase de cualquiera de las formas anteriores y también sus operaciones, se tiene lo siguiente:

```
Semaforo s;            // construimos un semáforo
s.cambiar();           // bien, función miembro pública
s.Semaforo::cambiar(); // lo mismo pero con resolución de ámbito
Color c;               // ERROR, ¿dónde está «Color»?
Semaforo::Color c;     // ERROR, «Semaforo::Color» es privado
s.c = Semaforo::ROJO;  // ERROR, «s.c» y «Semaforo::ROJO» privados
```

4.2.2. Miembros de datos y funciones miembro

El concepto de miembro de datos se corresponde con el de «atributo» y el de función u operador miembro con el de «método»[1].

Al definir una clase, los miembros de datos pueden ser de cualquier tipo válidamente definido. Esto excluye el propio tipo de la clase que se está definiendo, por estar incompleto (igualmente ocurría con las estructuras en C); no obstante, es posible emplear punteros o referencias al tipo.

Los miembros de datos pueden declararse constantes. Cuando un miembro de datos es `const` se está indicando que su valor, establecido durante la construcción del objeto, no podrá ser modificado con posterioridad. Esto ocurre independientemente de que el objeto se defina o no como una constante.

A veces es interesante que un objeto `const` pueda modificar alguno de sus miembros de datos (por ejemplo, como efecto colateral de una de sus operaciones observadoras). Para lograr esto se especificará en la clase que tales

[1] Tanto es así, que muchos autores emplean exclusivamente los términos «atributo» y «método», por ser generales de la programación orientada a objetos.

miembros de datos tienen almacenamiento `mutable`; esta palabra reservada indica que el miembro de datos nunca será constante, ni aunque el objeto al que pertenece lo sea.

Las funciones miembro se pueden definir interna o externamente a la definición de la clase. Una función miembro definida aprovechando su propia declaración dentro de la clase es automáticamente `inline` sin necesidad de especificar la palabra reservada.

Ejemplo:

Se puede definir el método *Semaforo::cambiar()* aprovechando su declaración dentro de la clase:

```
class Semaforo {
public:
  // ...
  void cambiar() { c = static_cast<Color>((c + 1) % (AMBAR + 1)); }
private:
  enum Color { ROJO, VERDE, AMBAR };
  Color c;
  // ...
};
```

Esto equivale a colocar en el exterior la siguiente definición `inline`:

```
inline void Semaforo::cambiar()
{
  c = static_cast<Color>((c + 1) % (AMBAR + 1));
}
```

Las funciones miembro pueden ser constantes, para indicar que corresponden a métodos observadores que se limitan a informar sobre el estado interno de un objeto sin alterarlo. Para ello, se emplea la palabra reservada `const` tras la lista de parámetros formales, tanto en la declaración como en la definición de la función.

A un objeto constante no se le puede aplicar directamente una función miembro que no lo sea. Esto es análogo a lo que ocurre cuando se intenta pasar un objeto constante a una función a través de un parámetro formal que no lo es[2].

[2]Por supuesto, se puede forzar a que se haga mediante un `const_cast` y bajo la responsabilidad del programador.

Cuando una función miembro `const` se aplica a un objeto, ésta se comporta como si todos sus miembros, a excepción de los especificados como `mutable`, fueran `const`. Es decir, como si el objeto en sí fuera constante.

Ejemplo:

A la clase *Semaforo* anteriormente presentada se le desea añadir una operación observadora, *Semaforo::comprobar()*, que efectúe un procedimiento rutinario de comprobación del estado del semáforo.

Esta comprobación no implica realmente cambios en el estado lógico del semáforo, aunque será necesario que modifique un nuevo miembro de datos, *revisado*, que indicará que el semáforo ha pasado la revisión con éxito. Para lograrlo, se especifica que éste es `mutable`.

```
class Semaforo {
public:
  // ...
  void comprobar() const;
  void cambiar();
private:
  enum Color { ROJO, VERDE, AMBAR };
  Color c;
  mutable bool revisado; // ¿ha pasado la inspección?
};
```

Se dice en este caso, que *Semaforo::comprobar()* es una función constante *desde el punto de vista lógico*, ya que, físicamente, altera el objeto.

4.2.3. Miembros estáticos

Cada objeto de una clase tiene un espacio en memoria reservado para cada miembro de datos. No obstante, a veces es conveniente que un dato sea compartido por todos los objetos de una clase, debiendo existir en memoria únicamente una copia.

Esto podría lograrse fácilmente representando dicho dato mediante una variable global, pero esto no es una buena idea[3] si su existencia sólo tiene sentido dentro de la clase.

La solución está en declararlo como un *miembro de datos estático*. Para ello se emplea la palabra reservada `static`. Los miembros de datos estáticos

[3]Rara vez el empleo de variables globales es buena idea.

se denominan «atributos de clase», puesto que representan propiedades de las clases y no de sus objetos individuales. La definición, acompañada de inicialización, se realizará fuera de la clase mediante resolución de ámbito (no es posible inicializarlos en la declaración, salvo en el caso excepcional de las constantes enteras).

Una aplicación usual de este tipo de miembros se da en simulación, ya que, a menudo, es necesario generar estadísticas sobre los objetos de una clase (el número de objetos activos, los que nacieron, los que murieron, etc.). Esta información, claramente, pertenece a la clase en sí, no a sus objetos individuales.

Igualmente, a veces es necesario emplear una determinada función auxiliar que no afecta al estado interno de los objetos y que sólo tiene sentido dentro de la labor que realiza una determinada clase.

La solución, de nuevo, está en declarar tal función como una *función miembro estática*. Análogamente, esto se consigue empleando la palabra reservada `static`.

Una función miembro estática está dentro del ámbito de la clase y, por lo tanto, puede acceder tanto a su parte pública como privada (al igual que cualquier función miembro), pero no a aquellos miembros que no sean estáticos, ya que son propios de los objetos individuales, a los que estas funciones no tienen acceso.

A los miembros estáticos de una clase se puede acceder directamente (resolviendo el ámbito, si es necesario) o a través de uno de sus objetos. Esta última forma no se aconseja, ya que, en realidad, enmascara el hecho de que dichos miembros pertenecen a la clase y no a ningún objeto concreto.

Ejemplo:

A continuación se modela, de forma muy simplificada, una entrada para el cine. Básicamente, ésta tiene un número de sesión, un número de asiento y un precio.

Sin embargo, para asignar el número de asiento a cada entrada de forma correlativa para cada sesión es útil disponer de cuál será el próximo número de asiento libre. Ésta no es una propiedad de cada entrada individual sino de todas las entradas de una misma sesión y por lo tanto se puede emplear un atributo de clase.

entrada/entrada.h

```
1 #ifndef ENTRADA_H_
```

```
2  #define ENTRADA_H_
3
4  class Entrada {
5  public:
6    void crear(int s);
7    void imprimir() const;
8    static double tarifa(int s);
9  private:
10   static double euros(double pta);
11   static int proximo_asiento[];
12   int sesion,
13       asiento;
14   double precio;
15 };
16
17 #endif
```

La implementación de la clase se muestra a continuación. Algunas de sus funciones son tan exiguas que bien podrían haberse definido «en línea» dentro de la cabecera.

entrada/entrada.cpp

```
1  #include <iostream>
2  #include <iomanip>
3  #include "entrada.h"
4  using namespace std;
5
6  // El núm. inicial de asiento será el 1 para las cuatro sesiones.
7
8  int Entrada::proximo_asiento[4] = { 1, 1, 1, 1 };
9
10 // Creación de una entrada (núm. de sesión entre 1 y 4)
11
12 void Entrada::crear(int s)
13 {
14   sesion = s - 1;
15   asiento = proximo_asiento[sesion]++;
16   precio = tarifa(sesion);
17 }
18
19 // Impresión de una entrada.
20
21 void Entrada::imprimir() const
22 {
23   cout << "Sesión " << sesion + 1
24        << "\tAsiento Nº " << asiento
25        << "\t" << setprecision(0) << precio << " pta"
```

```
26          << " (" << setprecision(2) << euros(precio) << " EUR)"
27          << endl;
28  }
29
30  // Cálculo de la tarifa en pesetas.
31
32  double Entrada::tarifa(int s)
33  {
34    // ... por ahora tarifa única ...
35    return 600.0;
36  }
37
38  // Función auxiliar de conversión de pesetas a euros.
39
40  double Entrada::euros(double pta)
41  {
42    return pta / 166.386; // 1 EUR = 166,386 pta
43  }
```

Nótese cómo se define *proximo_asiento*: inicializándolo externamente mediante resolución de ámbito. La palabra `static` no aparece en la definición.

Análogamente se definen las funciones miembro auxiliares *Entrada::tarifa()* y *Entrada::euros()*. Tampoco aparece la palabra `static` acompañando a las definiciones; simplemente se declaran `static`.

Al ser pública la primera, podría llamarse desde el exterior de la clase (resolviendo el ámbito, claro) para consultar la tarifa sin tener que generar una entrada:

```
double p = Entrada::tarifa();
```

Por el contrario, la segunda es privada y sólo puede ser empleada por las funciones miembro de la clase *Entrada*, en este caso, *Entrada::imprimir()*.

4.2.4. Sobrecarga de funciones miembro

Las funciones (y operadores) miembro se pueden sobrecargar al igual que las funciones externas a las clases. El único requisito, al igual que antes, es que no existan dos funciones bajo el mismo ámbito con idéntica signatura.

Sobrecarga de operadores miembro

La sobrecarga de operadores miembro merece algunos comentarios pese a que, sintácticamente, su empleo es el mismo que el de los operadores sobre-

cargados que puedan definirse fuera de la clase. Existe una sutil diferencia entre sobrecargar un operador como miembro de una clase y hacerlo fuera de ella.

Un operador miembro recibe implícitamente la dirección de un objeto, al igual que las restantes funciones miembro. Esto hace que, durante su declaración, su número efectivo de operandos disminuya en uno. Si el operador es unario, se declarará sin parámetros[4]; si es binario, con uno solo. Esto afecta a las conversiones implícitas, que *no se realizarán sobre el primer operando.*

Ejemplo:

La clase *Matriz* que aparece a continuación define operadores miembro unarios y binarios.

```
class Matriz {
public:
  // ...
  Matriz operator +();                // unario
  Matriz operator -();                // unario
  Matriz operator +(const Matriz& b); // binario
  Matriz operator -(const Matriz& b); // binario
  Matriz operator *(const Matriz& b); // binario (prod. matricial)
  Matriz operator *(double k);        // binario (prod. escalar)
  // ...
};
```

El empleo de estos operadores se realiza mediante la sintaxis habitual; por ejemplo:

```
Matriz r, a, b;

r = a + b;   // equivale a r = a.operator +(b);
r = - a * b; // equivale a r = a.operator -().operator *(b);
r = a * - b; // equivale a r = a.operator *(b.operator -());
```

Así, si en la primera asignación *a* no fuera un objeto de tipo *Matriz* sino de otro convertible implícitamente a él, la conversión no se llevaría a cabo; esto no ocurriría en el caso de *b*.

Si, en cambio, los operadores no fueran miembros de la clase se haría:

```
class Matriz {
  // ...
```

[4]Salvo en el caso de los operadores posfijos de incremento y decremento, que recibirán un parámetro «fantasma» de tipo `int` para distinguirse de sus versiones prefijas.

```
};

Matriz operator +(const Matriz& a);
Matriz operator -(const Matriz& a);
Matriz operator +(const Matriz& a, const Matriz& b);
Matriz operator -(const Matriz& a, const Matriz& b);
Matriz operator *(const Matriz& a, const Matriz& b);
Matriz operator *(const Matriz& a, double k);
```

y el significado cambiaría:

```
r = a + b;   // equivale a r = operator +(a, b);
r = - a * b; // equivale a r = operator *(operator -(a), b);
r = a * - b; // equivale a r = operator *(a, operator -(b));
```

Ahora, si fuera necesario realizar conversiones implícitas, éstas se aplicarían uniformemente en ambos operandos.

Además, así es posible sobrecargar el producto escalar para hacerlo conmutativo:

```
inline Matriz operator *(double k, const Matriz& a)
{
  return a * k;
}
```

cosa que dentro de la clase *Matriz* no se puede hacer.

En general, es preferible sobrecargar los operadores de carácter matemático de manera que no pertenezcan a la clase. Esto evita la falta de uniformidad que produce el hecho de que la aplicación de conversiones implícitas no sea simétrica y permite la conmutatividad de las operaciones matemáticas externas (aquéllas que se realizan sobre elementos de conjuntos distintos).

> Los operadores *operator* `=`, *operator* `[]`, *operator* `()` y *operator* `->` deben tener como primer operando un valor-i. Sólo pueden ser sobrecargados como funciones miembro (no estáticas, por supuesto).

Los operadores *operator* `=` (operador de asignación), *operator* `&` (operador de dirección) y *operator* `,` (operador coma) poseen significados predefinidos cuando se aplican a objetos: asignación miembro a miembro de un objeto a otro, cálculo de la dirección de un objeto y evaluación en secuencia, respectivamente. Si se desea modificar su comportamiento deben sobrecargarse. También puede prohibirse su empleo con objetos de una clase declarándolos privados en dicha clase; esto impide que se pueda acceder a ellos.

Sobrecarga `const`

Las funciones miembro constantes incluyen el especificador `const` como parte de su signatura; esto significa que es posible disponer de dos versiones idénticas de la función: una con `const` y otra sin él.

Ahora bien, ¿qué interés podría existir en esto? Por ejemplo, a veces una función devuelve una referencia a un miembro de datos del objeto sobre el que actúa (o al propio objeto) y puede emplearse de dos formas: para observar el objeto y para modificarlo a través de la referencia devuelta. Sin embargo, esto impide a priori su aplicación a objetos constantes. En este caso, puede desearse definir una sobrecarga `const` de la función; así existirán las dos versiones y el compilador elegirá la adecuada en función del objeto empleado.

Ejemplo:

La clase *Alumno* que se presenta a continuación contiene varias funciones miembro que permiten acceder directamente a determinadas partes del estado interno de sus objetos.

alumno/alumno.h

```
1  #ifndef ALUMNO_H_
2  #define ALUMNO_H_
3
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  class Alumno {
9  public:
10   // ...
11   string& nombre();
12   const string& nombre() const;
13   unsigned long int& dni();
14   unsigned long int dni() const;
15   // ...
16   void mostrar() const;
17 private:
18   string nombre_;
19   unsigned long int dni_;
20   // ...
21 };
22
23 // Acceso al nombre del alumno
```

```
24
25 inline string&       Alumno::nombre()       { return nombre_; }
26 inline const string& Alumno::nombre() const { return nombre_; }
27
28 // Acceso al DNI del alumno
29
30 inline unsigned long int& Alumno::dni()       { return dni_; }
31 inline unsigned long int  Alumno::dni() const { return dni_; }
32
33 // Presentación de los datos de un alumno
34
35 inline void Alumno::mostrar() const
36 {
37   cout << "Nombre: " << nombre() << endl
38        << "DNI:    " << dni()    << endl
39        // ...
40        ;
41 }
42
43 #endif
```

La primera versión de la función miembro *Alumno::nombre()* permite leer y modificar el nombre de un alumno que no haya sido definido como un objeto constante; la segunda permite únicamente leer el nombre de un alumno, ya que devuelve una referencia constante.

Análogamente, la primera versión de función de *Alumno::dni()* permite leer y modificar el DNI de un alumno, siempre que no se haya definido constante, mientras que la segunda permite únicamente leerlo, ya que no devuelve una referencia.

Alumno::mostrar() puede aplicarse tanto a objetos que sean constantes como a aquéllos que no lo sean, comprobándose que éstos no sean modificados por la aplicación de funciones que no sean observadoras. Por lo tanto, dentro de ella las llamadas a las funciones miembro *Alumno::nombre()* y *Alumno::dni()* corresponden a sus versiones `const`; de ahí su necesidad.

alumno/prueba.cpp

```
1 #include "alumno.h"
2
3 int main()
4 {
5   Alumno p1;
6   p1.nombre() = "Juan España";
7   p1.dni() = 51873029;
8   p1.mostrar();
```

```
9     const Alumno p2 = p1;
10    p2.mostrar();
11  }
```

Si se intentara hacer algo como:

```
p2.dni() = 51873026;
```

se produciría un error, ya que *p2* es un objeto constante, se emplearía la versión `const` de *Alumno::dni()* y ésta devolvería algo que no es un valor-i.

4.2.5. Clases y funciones amigas

A veces existe una relación de dependencia entre dos clases y es conveniente que exista un mecanismo por el que una clase pueda acceder a la parte privada de otra. Esto se consigue declarando que la clase que desea acceder a la otra es «amiga» suya mediante la palabra reservada `friend`.

```
class A {
public:
  friend class B;          // la clase «B» es amiga de «A»
  // ...
private:
  int d;
  // ...
};

class B {
public:
  void f(A& a) { ++a.d; } // modifica «a.d», que es privado
  // ...
};
```

Esto mismo ocurre con funciones y operadores. El caso más común es la sobrecarga de los operadores aritméticos binarios y de los operadores de inserción y extracción. A veces conviene que accedan directamente a la parte privada de la clase con la que trabajan. Sólo hay que realizar una declaración `friend` dentro de la clase de la función u operador que quiere acceder a su representación interna.

```
class Natural {
public:
  // ...
```

```
  friend Natural operator +(const Natural& a, const Natural& b);
  // ...
private:
  typedef unsigned char byte;
  valarray<byte> digito;
};

// Declaración externa del operador de suma sobrecargado
Natural operator +(const Natural& a, const Natural& b);
```

Una declaración `friend` puede colocarse indistintamente en la parte pública de la clase o en la privada; no importa dónde.

La relación de amistad entre dos clases, o entre una función u operador y una clase, permite pasar por alto el mecanismo de ocultación de datos, por lo que debe ser empleada con suma prudencia, ya que aumenta el acoplamiento entre los componentes relacionados.

4.3. Objetos

La raíz de los cambios de C++ respecto a C está en la inserción de funciones dentro de las estructuras; esto induce a pensar en éstas como conceptos. En C, una estructura es una aglomeración de datos, una manera de empaquetarlos de forma que se puedan tratar agrupadamente. Pero sólo eso, una mera conveniencia para el programador. Las funciones que operan sobre esas estructuras pueden estar diseminadas por el programa.

En cambio, con esas funciones dentro del paquete de la estructura, ésta se convierte en una nueva criatura, capaz no sólo de describir características (datos, como en C) sino comportamientos. Así surge el concepto de objeto, una entidad del programa que puede «recordar» (datos, atributos) y «actuar» (funciones, métodos).

Una vez vistas las clases, se puede considerar un objeto como un ejemplar de una de ellas. Bajo el punto de vista del compilador, sin embargo, un objeto no es más que una variable, una región de almacenamiento. Es un sitio donde uno puede guardar valores, y está implícito que también se pueden efectuar operaciones sobre esos datos.

4.3.1. El puntero `this`

Cuando en C se define una estructura (`struct`), las funciones que operan sobre ellas recibirán, además del resto de parámetros necesarios, un pará-

metro extra que es del tipo estructura sobre la que trabajarán, o, mejor, un puntero, quizá constante, de ese tipo estructura.

Ejemplo:

```
/* En fecha.h */

typedef struct Fecha {
  int dia, mes, anno;
} Fecha;

void asignar_fecha(Fecha *f, int d, int m, int a);
void mostrar_fecha(const Fecha *f);

/* En fecha.c */

void asignar_fecha(Fecha *f, int d, int m, int a)
{
  f->dia = d;
  f->mes = m;
  f->anno = a;
}
void mostrar_fecha(const Fecha *f)
{
  printf("%d/%d/%d\n", f->dia, f->mes, f->anno);
}

/* En prueba.c */

Fecha f;

asignar_fecha(&f, 12, 1, 2000);
mostrar_fecha(&f);
```

En C++, los métodos o funciones miembro no tienen ese parámetro del tipo puntero a la estructura o clase. ¿Cómo acceder desde dentro del método al objeto que lo ha llamado?

Ejemplo:

```
// En fecha.h

class Fecha {
  int dia, mes, anno;
public:
```

```
  void asignar(int d, int m, int a) {
    dia = d; mes = m; anno = a;
  }
  void mostrar() const {
    cout << dia << '/' << mes << '/' << anno << endl;
  }
};

// En prueba.cpp

Fecha f;
f.asignar(12, 1, 2000);
f.mostrar();
```

Como se ve al comparar el código C con el C++, los métodos en C++ tienen un parámetro menos. Dentro de ellos, los atributos se nombran tal cual (*dia*, *mes*, *anno*); el compilador, al ver un identificador sin cualificar en un método, busca primeramente en el ámbito de la clase y encuentra que *dia*, por ejemplo, está declarado en ella.

Asimismo se observa que en la llamada a un método no hace falta pasar como parámetro el objeto sobre el que actúa, puesto que está claro cuál es, ya que se escribe antes del nombre del método separado de éste por el operador `.` (*punto*)[5].

Los primeros compiladores de C++ no eran tales, sino que eran traductores de código C++ a código C. Producían un fichero en C que se compilaba entonces con el compilador nativo de C para obtener el ejecutable. Es fácil imaginar la tarea de este traductor en este caso comparando los dos ejemplos anteriores; el código C generado al traducir el ejemplo anterior en C++ sería muy parecido al del mismo ejemplo en C.

Actualmente hay compiladores nativos de C++, pero hacen «secretamente» algo parecido a lo de los primeros traductores: añaden automáticamente un primer parámetro invisible a cada método que es un puntero constante a la clase, conteniendo la dirección del objeto que llama al método en cuestión.

Este puntero constante, si hiciera falta utilizarlo dentro del cuerpo del método, se podría referenciar con la palabra reservada `this` (en inglés, *éste*).

De igual forma, el compilador añade automáticamente `this->` delante de cada referencia a un atributo de la clase, de manera que, por ejemplo, las definiciones de los métodos anteriores quedarían, tal como podría escribirlas en C el compilador, así:

[5] O un puntero al objeto, separado del método por el operador `->` (*flecha*).

```
// ¡Esto es ficticio! Ocurre «como si» el compilador
// C++ generara este código C

void asignar(Fecha* const this, int d, int m, int a)
{
  this->dia = d;
  this->mes = m;
  this->anno = a;
}
void mostrar(const Fecha* const this)
{
  cout << this->dia << '/' << this->mes
       << '/' << this->anno << endl;
}
```

y las llamadas serían así:

```
Fecha* f;
asignar(&f, 12, 1, 2000);
mostrar(&f);
```

Como se ve, la definición de `this` para una clase X sería `X* const this`; esto es, un puntero constante: `this` no es modificable, no puede hacerse apuntar a otro objeto[6].

Además, en funciones `const`, como es el caso de *mostrar()* en el ejemplo, la declaración sería `const X* const this`; o sea, un puntero no modificable que apunta a un objeto no modificable. Es decir, no sólo no se puede hacer que `this` apunte a otro sitio, sino que a través de él tampoco podemos modificar el objeto; en otras palabras, tanto `this` como `*this` serían constantes.

Por último, los métodos estáticos (declarados `static`) no pertenecen a un objeto en particular sino a toda la clase, son como funciones normales pero con ámbito de clase. Por lo tanto, ellas no reciben este parámetro implícito, dentro de ellas no puede emplearse `this`; o lo que es lo mismo, no se puede acceder directamente a atributos, salvo a los estáticos, que son como variables globales con ámbito de clase.

Normalmente no necesitamos emplear explícitamente `this` dentro del cuerpo de un método, aunque podemos poner `this->dia` en vez de simplemente `dia`; pero pronto veremos situaciones donde el empleo de `this` será necesario. De todas formas es muy importante entender el concepto subyacente a este puntero «invisible».

[6] Pero antiguamente, en los primeros compiladores, esto no era así.

4.3.2. Construcción y destrucción

En la programación de hoy día, la reutilización de código es crucial. Por eso es muy importante la realización de bibliotecas. En C, las bibliotecas eran principalmente de funciones. El C++ mejora ampliamente el empleo de bibliotecas tomando todos los componentes de una bibilioteca típica de C y encapsulándolos en una estructura: un tipo abstracto de datos, una clase de aquí en adelante.

Además, la ocultación de datos impone límites sobre lo que el usuario programador podrá manipular: los mecanismos internos de operación de un tipo de datos están sólo bajo el control y discreción de quien lo diseñó.

Pero aún hay más. Un gran número de errores que tienen lugar programando en C se cometen porque el programador olvida inicializar o «limpiar» (poner a cero) una variable. Esto ocurre especialmente en bibliotecas, cuando los usuarios no saben cómo inicializar una `struct`, o ni siquiera saben que deben hacerlo.

En C++ el concepto de inicialización y limpieza es esencial para mantener simple el empleo de bibliotecas y para eliminar los errores que tienen lugar cuando uno se olvida de esas actividades.

Inicialización garantizada con el constructor

Podemos diseñar una clase que tenga muchos atributos y métodos útiles; cuando definamos un objeto de ella, ¿qué estado tendrá, cuáles serán los valores de sus atributos, cómo inicializamos ese objeto?

La solución obvia es tener un método de nombre *inicializar()* o parecido, que se encargue de ello. Similarmente, tendríamos otro método *limpiar()* o *borrar()* que se encargaría de la tarea contraria.

Claramente, este método es ineficaz y propenso a errores. El programador se ve forzado a llamar a *inicializar()* justo entre la definición del objeto y su primer uso; si se le olvida, el objeto estará en un estado inconsistente.

En C++, la inicialización es demasiado importante como para dejarla en manos del usuario. Por eso el diseñador de la clase puede garantizar la correcta inicialización de cada objeto definiendo una función miembro especial llamada *constructor*. Si una clase tiene constructor, el compilador lo llamará automáticamente cada vez que se cree un objeto, en el punto de la definición, antes de que el usuario pueda usarlo aún. Esto no es una opción para el usuario, que no puede impedirlo.

¿Qué nombre darle a esta función especial, al constructor? Tenía que ser un

nombre que no pudiera coincidir con cualquier otro que a uno se le pudiera ocurrir como miembro de la clase. Y como es el compilador el que llama a esta función, él tiene que conocer su nombre. La solución escogida por Stroustrup es la más fácil y lógica: el nombre del constructor es justamente el nombre de la clase.

Ejemplo:

Aquí hay una clase simple con un constructor:

```
class X {
  int i;
public:
  X();     // constructor
};
```

Ahora, cuando se define un objeto:

```
void f()
{
  int a;
  a = 4;
  cout << a;
  X x;
  // ...
}
```

ocurre lo mismo que para los tipos básicos, como a: se reserva memoria para almacenar el objeto x. Observe que esto ocurre seguramente en la entrada en el bloque de la función; puesto que a y x son variables automáticas, el compilador reserva memoria para ellas en la pila de la función, y seguramente esto lo hace muy rápidamente simplemente «moviendo hacia abajo»[7] el puntero de pila; pero es en el punto de la definición de x, tras mostrar el valor de a, cuando inserta código para llamar a la función *X::X()*: el constructor, puesto que antes de ese punto el identificador x no está disponible.

Como cualquier función, el constructor puede tener parámetros, para permitir especificar cómo construir un objeto, dándole valores iniciales. Esto garantiza la correcta inicialización a valores apropiados (el propio constructor debe asegurarse de que los valores suministrados sean realmente apropiados).

[7] Esto es relativo; el valor real del puntero de pila puede aumentar o disminuir, dependiendo de la máquina.

Como puede haber distintas formas de inicializar un objeto, puede haber varios constructores, y de hecho esto es lo normal: los constructores pueden y suelen sobrecargarse. Esto nos lleva a la existencia de varios tipos de constructores que se verán a continuación.

Y esto es casi todo respecto a los constructores: son funciones miembro con un nombre especial, el de la propia clase, que son llamadas automáticamente por el compilador cada vez que se crea un objeto. Sin embargo, la existencia de constructores elimina un gran número de problemas y hace el código más fácil de leer, sin funciones con nombres como *inicializar()* separadas de la definición. En C++, definición e inicialización son conceptos unificados para objetos definidos por el usuario: no hay una sin la otra.

Una cosa más: los constructores, y el destructor, que estamos a punto de ver, son funciones especiales en cuanto que no tienen valor de devolución. Esto es diferente a no devolver nada, es decir, que el tipo de retorno sea *void*; no, ni siquiera devuelven *void*. Los constructores y el destructor no devuelven nada, en efecto, pero uno no tiene la opción de que puedan devolver algo. Es decir, uno puede definirse una función que devuelva lo que uno quiera. Puede decidir que no devuelva nada (*void*), o que devuelva algo. Si los constructores o destructor tuvieran esta posibilidad, uno podría hacer que devolvieran algo distinto de *void*, lo cual estaría mal, porque el compilador tendría que averiguar qué hacer con el valor devuelto, o sería el usuario el que tendría que llamarlos explícitamente, lo que eliminaría parte de su utilidad.

Limpieza garantizada con el destructor

Un programador en C puede pensar en la importancia de la inicialización, pero rara vez en la de la limpieza; después de todo, ¿qué se necesita para limpiar un *int*? Nada, simplemente uno se olvida de él. Sin embargo, con bibliotecas de clases, no es tan seguro a veces el olvidarse de un objeto una vez utilizado. Éste podría haber escrito algo en la pantalla que habría que borrar, o podría haber reservado memoria dinámica que habría que liberar, o podría haber abierto ficheros que habría que cerrar, etc.

En C++, la limpieza es tan importante como la inicialización, y por eso se garantiza mediante una función especial llamada el *destructor*.

La sintaxis para el destructor es similar a la del constructor. El nombre es de nuevo el de la clase, pero para distinguirlo del constructor, se le antepone el signo de la tilde de la eñe o virgulilla: ~. Esto es peculiar porque rompe la regla de formación de identificadores, donde este carácter no está permitido; pero sin embargo, al ser ~ el operador de bits de complemento a 1, el nombre sugiere: «el complemento del constructor».

Además, el destructor nunca tiene ningún parámetro, porque la destrucción nunca necesita ninguna opción especial. Por tanto, sólo hay un destructor, no puede sobrecargarse, y su declaración sería algo así:

```
class X {
public:
  ~X();
  // ...
};
```

Si no se define el destructor para una clase, el compilador proporciona automáticamente uno, que no hace nada especial; para la clase anterior su definición sería:

```
X::~X() {}
```

El destructor es llamado automáticamente por el compilador cuando un objeto termina su vida, o sale fuera de ámbito. Uno puede figurarse cuándo se llama a un constructor viendo la definición de un objeto, pero la única evidencia de llamada del destructor es la llave de cierre de un bloque. Incluso se llama al destructor cuando una instrucción `goto` u otra de ruptura de control lleve el flujo del programa fuera del ámbito de un objeto.

> Pero un salto no local, que se hace mediante las funciones y macros de la biblioteca estándar de C *setjmp()* y *longjmp()*, no provoca la llamada del destructor, según el Estándar, aunque algún compilador pueda hacerla.

Construcción y destrucción en memoria dinámica

Supóngase que se quiere construir un objeto en memoria dinámica, y más tarde destruirlo, y que quisiéramos utilizar solamente las funciones de la biblioteca estándar de C *malloc()* y *free()*.

```
class X {
  int i;  // un atributo privado
public:
  X();    // un constructor
  ~X();   // el destructor
  // ...  // el resto de los miembros
};

#include <cstdlib>
#include <cassert>
```

```
using namespace std;

void f()
{
  X* px = static_cast<X*>(malloc(sizeof(X)));
  assert(px);  // comprobamos que px != 0
  // ... emplear *px ?
  free(px);
}
```

¿Qué tenemos en *px*? La dirección de una zona de memoria de tamaño suficiente para almacenar un objeto de tipo X, pero ¿qué hay en esa zona? *malloc()* no pone nada en ella; se dice que hay *basura*; o sea, lo que hubiera allí antes, ciertamente desconocido para nosotros. En otras palabras, **no se ha llamado al constructor y, además, no tenemos forma de hacerlo**. De igual modo, al liberar la memoria con *free()*, no se llama al destructor ni hay forma de llamarlo. En definitiva, no tenemos un objeto en sentido estricto; sólo una zona de memoria reservada con el tamaño correcto.

Está claro pues que no es ésta la forma de trabajar en C++: las funciones de C de memoria dinámica no nos sirven a la hora de trabajar con objetos definidos por nosotros. Precisamente por eso se inventaron los operadores `new` y `delete`.

El operador `new` hace lo siguiente:

1. calcula el tamaño de memoria que tiene que pedir.
2. llama a una función que reserve esa cantidad de memoria. Esa función es probablemente la propia *malloc()*, pero no es seguro y además podemos modificar la forma en que `new` obtenga memoria, sobrecargando `new`.
3. se asegura de que se ha obtenido la memoria requerida. Si no es así:
   a) devuelve 0 si se le pasó el parámetro especial *nothrow*.
   b) si hemos definido e instalado una función para manejar el caso de falta de memoria, la llama.
   c) si no ocurre lo anterior, lanza una excepción estándar, *bad_alloc*.
4. **llama al constructor del objeto**, pasándole implícitamente la dirección de memoria obtenida como `this`.
5. devuelve la dirección de la memoria obtenida, transformada al tipo puntero apropiado.

Análogamente, el operador `delete` hace lo siguiente:

1. si se le pasa el puntero nulo, no hace nada.
2. si se le pasa la dirección de una zona de memoria (que debería haber sido reservada por `new`), **llama al destructor del objeto**, pasándole esa dirección como `this`.
3. libera la memoria, quizás llamando a la función *free()*, aunque no es seguro, y podemos variar esto sobrecargando este operador.

Por eso en C++ estos operadores no son sólo una conveniencia frente a *malloc()*/*free()*, sino que son indispensables. El código anterior quedaría:

```
void f()
{
  X* px = new X;  // realmente _construye_ el objeto *px
  // ... empleamos *px
  delete px;      // destruye *px y libera la memoria
}
```

El constructor predeterminado

Es el que puede llamarse sin parámetros. Esto puede suceder porque realmente no tenga ninguno, o porque todos tengan valores prefijados.

El constructor predeterminado se suele utilizar para crear un objeto al que se darán valores más adelante por otros medios, o bien para crear un objeto en su estado más «natural». Pero es indispensable su existencia si queremos crear un vector (de bajo nivel) de objetos. Hasta tal punto que si no se define **ningún** otro constructor, el compilador proporciona uno predeterminado automáticamente, que no hace nada especial. Su definición para una clase X sería algo así:

```
X::X() {}
```

Esto es esencial para mantener la compatibilidad con el lenguaje C; si el compilador no proporcionara este constructor predeterminado automáticamente, no se podrían definir vectores de bajo nivel de `struct`s como se hace en C.

Ejemplo:

Supongamos una clase para representar a los puntos del plano.

```
class Punto {
public:
  Punto(double x, double y) { x_ = x; y_ = y; }  // constructor
  // ...
private:
  double x_, y_;
};

// ...

Punto a(3., 4.);          // Bien, se llama a a.Punto(3.0, 4.0)
Punto* p = new Punto(0., 0.);      // Bien, en memoria dinámica
Punto b;                  // ERROR, no hay ctor. predeterminado
Punto* pp = new Punto;    // ERROR, no hay ctor. predeterminado
Punto* vp = new Punto[5]; // ERROR, no hay ctor. predeterminado
Punto c[5];               // ERROR, no hay ctor. predeterminado
```

El compilador no ha proporcionado un constructor predeterminado puesto que el programador ha expresado su deseo de cómo construir objetos *Punto* escribiendo su propio constructor.

Si no se hubiera escrito el constructor anterior, el compilador habría proporcionado uno predeterminado y las cuatro declaraciones últimas hubieran sido válidas (pero no las dos primeras, claro está).

La alternativa obvia es escribir el constructor predeterminado además del ya existente. Como ya se ha dicho, es habitual sobrecargar los constructores:

```
class Punto {
public:
  Punto() { x_ = y_ = 0.0; }  // constructor predeterminado
  Punto(double x, double y) { x_ = x; y_ = y; } // el otro
  // ...
```

Con esto todas las declaraciones anteriores serían válidas.

Sin embargo, en este ejemplo podemos simplificar algo; gracias a los parámetros predeterminados de las funciones en C++, podemos disimular la sobrecarga escribiendo un solo constructor que valga de constructor predeterminado, de uno o de dos parámetros. Es como si hubiera tres constructores en vez de uno. Para variar, escribiremos ahora la definición aparte (como se debería en un programa real).

```
class Punto {
public:
  Punto(double x = 0.0, double y = 0.0);
  // ...
};
```

```
inline Punto::Punto(double x, double y)
{
  x_ = x;
  y_ = y;
}
```

Con esto, también todas las declaraciones anteriores serían válidas, y además podríamos crear objetos con un solo parámetro:

```
Punto e(10.5);    // e.Punto(10.5, 0.0);
Punto f = 10.5;   // equivale a lo anterior
```

Lista de inicializadores

A veces las inicializaciones de los atributos de una clase no pueden expresarse dentro del código de un constructor; otras veces se puede pero no es eficiente.

Ejemplo:

```
class Persona {
  string nombre;
  const int edad;
public:
  Persona(const char* n, int e);
  // ...
};

inline Persona::Persona(const char* n, int e)
{
  nombre = n; // asignación
  edad = e;   // ERROR, ¡«edad» es const!
}
```

Como se ve, dentro del código hay asignaciones, no inicializaciones. Y a un objeto `const` no se puede asignar nada; más bien hay que inicializarlo, por tanto la segunda línea es un error de compilación.

Y la primera no lo es, pero la asignación del *string* desde una cadena de bajo nivel implica primeramente la construcción del subobjeto *string* vacío, con su constructor predeterminado, y luego la construcción de un objeto temporal que se copia en *nombre*; algo así:

```
string nombre;
nombre = string(n);
```

que equivale a

```
string nombre;  // construcción del subobjeto string
string temp(n); // construcción de un string temporal
nombre = temp;  // asignación entre strings
temp.~string(); // destrucción del objeto temporal
```

La solución a estos problemas se llama *lista de inicialización del constructor* y consiste en una lista de inicializaciones de subobjetos (es decir, llamadas a sus constructores) separados por comas; entre el nombre de la clase, y separada de ella por el signo de dos puntos, y antes del cuerpo de la función.

El ejemplo anterior quedaría correctamente escrito así:

```
inline Persona::Persona(const char* n, int e): nombre(n), edad(e) {}
```

> Esto requiere evidentemente que un subobjeto se pueda construir con esta sintaxis funcional. Pero *e* es un simple *int*, no un objeto definido por el usuario, no una clase. ¿Acaso los tipos fundamentales tienen constructores?
>
> La respuesta es sí y no. No, no tienen constructores ni destructores, pero el compilador hace que, algunas veces, se comporten como si lo tuvieran. Esto es necesario por consistencia en el lenguaje, y para casos como éstos y otros que se dan con las plantillas, como se verá en su momento. Así, para todos los tipos fundamentales de C++ están definidos los «constructores» predeterminados y de copia (que vamos a ver a continuación), la asignación (evidentemente) y el «destructor». El «constructor» predeterminado inicializa el objeto a 0, el 0 apropiado al tipo de datos concreto.
>
> Ejemplo:
>
> ```
> int a = 3;
> int b(a);
>
> auto int c;
> auto int d = int();
> a.~int();
> b.int::~int();
> ```
>
> La primera línea es lo normal; *a* se inicializa con el valor 3; en la segunda inicializamos *b* con el valor de *a* pero empleando la notación de constructor, como si *b* fuera un objeto de alguna clase construida por nosotros. Esto equivale a

```
int b = a;
```

En cambio, cuando definimos *c* como una variable automática, no se llama al constructor predeterminado de *int*, puesto que *int* no es realmente un tipo definido por el usuario y realmente no posee constructores; es decir, *c* no vale 0, sino que contiene *basura* por ser una variable automática: está sin inicializar.

En la siguiente línea inicializamos *d* llamando explícitamente a su constructor predeterminado, que es `int::int()` o más abreviadamente `int()`; *d* vale 0.

Por último llamamos a los destructores de *a* y *b*, en sus formas corta y larga. Esto podría emplearse si quisiéramos destruir una variable antes de llegar a su fin de ámbito. (Pero cuidado: esto sólo debería hacerse, y muy raras veces, con tipos fundamentales, porque para los otros se volvería a llamar al destructor de nuevo al salir de su ámbito.)

Por supuesto estas formas de llamadas a constructores y destructores no se ven nunca en código real, por muy correctas que sean. Con tipos fundamentales, utilice siempre la forma «normal» como en C.

La sintaxis de inicializadores se pensó originalmente para la herencia, que se verá más adelante, pero es necesaria en casos donde los subobjetos son constantes o referencias, pues éstos no admiten más que inicialización; y además, como se ha visto, son muy convenientes en otros casos. Nuestro consejo es utilizar preferentemente esta sintaxis en los constructores siempre que se pueda.

4.3.3. El constructor de copia

El constructor de copia es el que puede recibir un único parámetro del mismo tipo de su clase.

El parámetro formal del constructor *debe* ser obligatoriamente una referencia al objeto de la misma clase. Normalmente será además una referencia constante, puesto que el objeto recibido por el constructor se va a copiar, no a modificar. No puede pasarse el parámetro real del constructor por valor, puesto que esto daría lugar a una recursión infinita: precisamente se va a definir con esta función el paso por valor. Tampoco puede pasarse un puntero, puesto que lo que se tiene que copiar es un objeto de la clase, no un puntero a ella. Por estos motivos, las referencias son algo imprescindible en C++, aunque en otros casos puedan verse como simple conveniencia.

Si uno no escribe el constructor de copia de una clase, el compilador proporciona uno automáticamente[8]. Esto es necesario para preservar la compatibilidad con C, donde las estructuras del mismo tipo (y recuerde que una clase

[8] A menos que la clase solo contenga miembros constantes o referencias.

no es más que una estructura con el acceso predeterminado privado en lugar de público) podían copiarse, pasarse y devolverse por valor.

Este constructor de copia proporcionado por el compilador llama a su vez recursivamente a los constructores de copia, si los hay, de los subobjetos que constituyan la clase en cuestión, y hace una copia miembro a miembro de los atributos.

Esto puede bastar para clases simples, pero conduce a errores en otros casos, en los que se deberá pues definir este constructor de la forma apropiada.

El constructor de copia se llama en los siguientes casos:

- se pasa un objeto a una función por valor.
- una función devuelve un objeto por valor.
- un objeto se inicializa a partir de otro.

Ejemplo:

```
class X;  // definiciones en otro sitio
X f(X);

X a;

X b = f(a);
f(a);

X c(b);
X d = a;
```

La función *f()* recibe un objeto de una cierta clase X por valor, y devuelve un objeto X de la misma forma. En la llamada `f(a)` el parámetro real a se copia mediante el constructor de copia al parámetro formal de la función. Ésta devuelve entonces un objeto de tipo X que se copia mediante el constructor de copia en b. ¿Qué hubiera pasado si hubiésemos escrito en su lugar esto?

```
X b;
b = f(a);
```

El objeto b ya está construido, y lo que devuelve *f()* se copia mediante el constructor de copia en un objeto temporal invisible que se asigna (veremos muy pronto cómo definir la asignación) a b. Inmediatamente después, este objeto temporal se destruye.

Algo análogo ocurre en la llamada a *f()* donde se desecha el resultado. Éste debe copiarse en algún sitio, para lo que se crea un objeto temporal que se destruye inmediatamente.

En las dos últimas líneas, los objetos *c* y *d* se crean inicializándose con copias de *b* y *a* respectivamente. Observe que las dos sintaxis son equivalentes. La última línea no es una asignación puesto que ahí *d* aún no existe y hay que crearlo. Se llama pues al constructor de copia.

4.3.4. Sobrecarga del operador de asignación

Relacionada con la copia de objetos está la asignación. Por eso cuando haya que definir el constructor de copia de una clase, casi forzosamente habrá que definir también cómo se quiere que sea la asignación. Para ello se definirá el operador de asignación como miembro de la clase (esto es obligatorio en este operador). Como ocurría con el constructor de copia, si no sobrecargamos el operador de asignación, el compilador se encargará de que ésta tenga lugar mediante una asignación miembro a miembro. Esto puede bastar para clases simples, pero no para otros casos. Veremos ahora por fin uno de éstos donde es necesario definir la copia de objetos de la clase.

Supongamos que tenemos que trabajar con datos de personas. Para simplificar, sólo consideraremos el nombre completo, la edad en años y la estatura en centímetros. Supondremos también que no tenemos disponible la biblioteca estándar de *string*s o que no sabemos utilizarla. Entonces guardaremos el nombre como una cadena de caracteres de bajo nivel, al estilo C, utilizando, eso sí, memoria dinámica para no desperdiciar espacio o no quedarnos cortos. Éste sería un esbozo de la clase sin definir la copia de objetos:

persona1.h

```
1  // Clase Persona, versión sin copia definida
2
3  #include <cstring>
4
5  class Persona {
6    char* nombre_;
7    int edad_, estatura_;
8  public:
9    Persona(const char* n, int ed, int est)
10   : edad_(ed), estatura_(est)
11   {
12     nombre_ = new char[std::strlen(n) + 1];
13     std::strcpy(nombre_, n);
14   }
```

```
15    ~Persona() { delete[] nombre_; }
16    void estatura(int e) { estatura_ = e; }
17    int estatura() const { return estatura_; }
18    void edad(int e) { edad_ = e; }
19    int edad() const { return edad_; }
20    char* nombre() const { return nombre_; }
21  };
```

Veamos ahora un programa de ejemplo que muestra el fallo:

```
1   // Prueba de la clase Persona sin copia definida
2   #include "persona1.h"
3   #include <iostream>
4   #include <clocale>
5   using namespace std;
6
7   void ver_datos(Persona p)
8   {
9     cout << p.nombre() << " tiene " << p.edad() << " años y mide "
10         << p.estatura() / 100.0 << " m." << endl;
11  }
12
13  int main()
14  {
15    if (!setlocale(LC_NUMERIC, "")) cerr << "setlocale(): error\n";
16    Persona pp("Pepe", 32, 180);
17    ver_datos(pp);
18    cout << '¡' << pp.nombre() << " ha sido destruido!" << endl;
19  }
```

Cuando se compila y se ejecuta este programa, se obtiene la salida siguiente:

```
Pepe tiene 32 años y mide 1,8 m.
¡H&@H&@re/l ha sido destruido!
```

Se crea un objeto *pp* de tipo *Persona*, donde se reserva memoria dinámica para almacenar una cadena de caracteres de C con el nombre, y se pasa por valor a la función *ver_ datos()*, donde se copia al parámetro real *p*. Como no se ha definido el constructor de copia, se usa el proporcionado por el compilador. Éste ha copiado simplemente el contenido del puntero *pp.nombre_* a *p.nombre_* , de forma que los dos apuntan a la misma zona de memoria. Al acabar la función, se llama al destructor de *Persona*, que libera esa zona de memoria, y como los dos objetos apuntaban a ella, ahora *p.nombre_* se ha quedado con la dirección de una zona de memoria no reservada. Al acceder

a ella mediante `pp.nombre()` se ve claro el fallo. No se ha copiado esa zona de memoria, y debería haberse hecho.

Para arreglar todo esto, definiremos un constructor de copia; y también el operador de asignación, pues esta operación padecerá el mismo problema:

Ejemplo:

```
Persona *p = new Persona("Pepe", 38, 180);
Persona q("Joaquín", 40, 175);
q = p;    // q == { "Pepe", 38, 180 }
          // ¡"Joaquín" queda reservado pero inaccesible!
delete p; // se destruye p, ¡pero también q.nombre_!
```

He aquí la nueva versión de la clase.

persona2.h

```
1  // Clase Persona, versión con copia definida
2
3  #include <cstring>
4
5  class Persona {
6    char* nombre_;
7    int edad_, estatura_;
8  public:
9    Persona(const char* n, int ed, int est)
10     : edad_(ed), estatura_(est)
11     {
12       nombre_ = new char[std::strlen(n) + 1];
13       std::strcpy(nombre_, n);
14     }
15   Persona(const Persona& otra)
16     : edad_(otra.edad_), estatura_(otra.estatura_)
17     {
18       nombre_ = new char[std::strlen(otra.nombre_) + 1];
19       std::strcpy(nombre_, otra.nombre_);
20     }
21   ~Persona() { delete[] nombre_; }
22   Persona& operator =(const Persona& otra)
23     {
24       if (this != &otra) {
25         delete[] nombre_;
26         edad_ = otra.edad_;
27         estatura_ = otra.estatura_;
28         nombre_ = new char[std::strlen(otra.nombre_) + 1];
29         std::strcpy(nombre_, otra.nombre_);
```

```
30        }
31        return *this;
32      }
33    void estatura(int e) { estatura_ = e; }
34    int estatura() const { return estatura_; }
35    void edad(int e) { edad_ = e; }
36    int edad() const { return edad_; }
37    char* nombre() const { return nombre_; }
38  };
```

Y el mismo programa de ejemplo de antes, ligeramente modificado:

```
1   // Prueba de la clase Persona con copia definida
2   #include "persona2.h"
3   #include <iostream>
4   #include <clocale>
5   using namespace std;
6
7   void ver_datos(Persona p)
8   {
9     cout << p.nombre() << " tiene " << p.edad() << " años y mide "
10         << p.estatura() / 100.0 << " m." << endl;
11  }
12
13  int main()
14  {
15    if (!setlocale(LC_NUMERIC, "")) cerr << "setlocale(): error\n";
16    Persona pp("Pepe", 32, 180);
17    ver_datos(pp);
18    cout << '¡' << pp.nombre() << " NO ha sido destruido!" << endl;
19  }
```

Cuando se compila y se ejecuta este programa, se obtiene la salida siguiente:

```
Pepe tiene 32 años y mide 1,8 m.
¡Pepe NO ha sido destruido!
```

Esta vez ha funcionado todo correctamente.

El constructor de copia reserva memoria para almacenar el nombre del objeto a copiar, y copia el nombre mediante la función *strcpy()* de la biblioteca estándar de C.

El operador de asignación hace algo parecido, pero es distinto del constructor porque el objeto donde se va a efectuar la copia ya existe. Primero, la línea

```
if (this != &otra)
```

impide que la auto-asignación sea destructiva; si `this`, que contiene la dirección del objeto destino de la copia, es igual a la dirección del otro objeto, el que se va a copiar, es que los dos son iguales: se estaría intentando una auto-asignación. Puede pensarse que es absurdo escribir alguna vez algo como

```
p = p;
```

pero por muy absurdo que sea, es C++ «legal», y no debe tener efecto ninguno; con esa comprobación hacemos que la auto-asignación sea inofensiva.

Después tenemos que liberar el espacio de memoria que ocupaba el objeto que va a sobreescribirse; si no, se quedaría reservado pero inaccesible. Esto no es un «fallo» apreciable pero es un desperdicio de memoria en ausencia de un sistema recolector automático de basura.

Después de esto, ya podemos reservar memoria, como se hacía en el constructor, y hacer la copia del nombre.

El operador de asignación podría haberse declarado como *void*; esto es, que no devolviera nada. Esto permitiría asignaciones simples como `p = q`; pero no algo como `p = q = r`, que equivale a `p = (q =r)`. Por ello, para emular mejor el comportamiento del *operator* `=`, se devuelve una referencia al propio objeto recién asignado, que es justamente `*this`.

Se ve pues que el constructor de copia y el operador de asignación tienen una parte común, así como el operador de asignación y el destructor; se podrían definir métodos privados (que por lo tanto sólo podrán llamar otros métodos de la clase solamente) que realizaran la tarea de copiar y destruir incondicionalmente, y así las definiciones del constructor de copia, del operador de asignación y del destructor quedarían más claras y seguirían siempre el mismo patrón, salvo en casos especiales:

persona3.h

```
1  // Clase Persona, versión con copia definida
2  // y con los métodos privados copiar() y destruir()
3
4  #include <cstring>
5
6  class Persona {
7   public:
8    Persona(const char* n, int ed, int est)
9      : edad_(ed), estatura_(est)
10     {
11       nombre_ = new char[std::strlen(n) + 1];
```

```
12        std::strcpy(nombre_, n);
13      }
14    Persona(const Persona& otra)
15      {
16        copiar(otra);
17      }
18    ~Persona() { destruir(); }
19    Persona& operator =(const Persona& otra)
20      {
21        if (this != &otra) {
22          destruir();
23          copiar(otra);
24        }
25        return *this;
26      }
27    void estatura(int e) { estatura_ = e; }
28    int estatura() const { return estatura_; }
29    void edad(int e) { edad_ = e; }
30    int edad() const { return edad_; }
31    char* nombre() const { return nombre_; }
32   private:
33    char* nombre_;
34    int edad_, estatura_;
35    void copiar(const Persona& otra)
36      { // copia incondicionalmente otro objeto
37        edad_ = otra.edad_;
38        estatura_ = otra.estatura_;
39        nombre_ = new char[std::strlen(otra.nombre_) + 1];
40        std::strcpy(nombre_, otra.nombre_);
41      }
42    void destruir()
43      {  // destruye incondicionalmente la memoria reservada
44        delete[] nombre_;
45      }
46  };
```

4.3.5. Conversiones

C++ sabe cómo convertir algunos tipos a otros sin necesidad de emplear los operadores de conversión vistos en §2.4.13 (conversiones *implícitas*). Algunos tipos de conversiones implícitas son:

Triviales permiten poner un tipo donde se requiera el otro; dos funciones sobrecargadas no pueden diferenciarse en estos tipos:

T	$\longleftrightarrow T$`&`
T`[]`	$\longleftrightarrow T$`*`
T o T`&`	$\longleftrightarrow$`const` o `volatile` T
T`()`	$\longleftrightarrow T$ `(*)()`
campo de bit	$\longleftrightarrow$el tipo correspondiente

Promociones también permiten poner un tipo donde se requiera el otro, pero se pueden definir funciones sobrecargadas que dependan de esta diferencia. Son conversiones entre tipos aritméticos a los correspondientes de mayor tamaño.

double	$\longrightarrow$*long double*
float	$\longrightarrow$*double*
char, *short*, *enum*	$\longrightarrow$*int*
int	$\longrightarrow$*long*

Estándares son un conjunto de conversiones muy diversas; producen ambigüedad en el momento de la llamada a funciones sobrecargadas que sólo difieran en estos tipos. Algunas son:

- De signo: entre dos tipos `signed` y `unsigned`.
- De punteros: de un puntero a cualquier tipo (excepto a funciones y a miembros de clase) a *void*`*`, y de cualquier puntero `const` o `volatile` al correspondiente sin estos modificadores.
- De puntero nulo: la constante 0 se convierte a cualquier tipo puntero (no hace falta la macro *NULL*).
- Aritméticas: entre tipos aritméticos, salvo las conversiones ya mencionadas. También hay que excluir el paso de un entero a un enumerado. La constante 0 se convierte a cualquier valor cero aritmético.
- Herencia: se verán en la sección §4.6.3.

Definidas por el usuario Cuando el usuario define una nueva clase, ¿cómo puede saber el compilador si debe o puede convertir entre un tipo y el de la nueva clase? Hay que decírselo, y a continuación se mostrará cómo hacerlo.

El constructor de conversión

Es el que puede llamarse con un solo parámetro de un tipo distinto al de la clase; proporciona una conversión desde el tipo del parámetro al de la clase.

Ejemplo:

La clase *NombreCaracter* servirá para mostrar cada carácter del código ISO-Latin1 con su representación:

nombrecaracter.h

```
1  // Nombres de los caracteres
2
3  class NombreCaracter {
4   public:
5    NombreCaracter(int c = 0) : valor(c % 256) {}
6    const char* nombre() const { return nombres[valor]; }
7   private:
8    int valor;
9    static const char* nombres[256];
10 };
```

El atributo estático hay que definirlo e inicializarlo fuera de la clase:

nombrecaracter.cpp

```
1  /* Nombres de los caracteres del código ISO-8859-1
2  */
3
4  #include "nombrecaracter.h"
5
6  const char* NombreCaracter::nombres[256] = {
7    "Null", // 0
8    "Start Of Heading",
9    // ...
10   "Tilde",
11   "Delete", // 127
12   "Not Assigned",
13   // ...
14   "NO-BREAK SPACE", // 160
15   "INVERTED EXCLAMATION MARK",
16   // ...
17   "LATIN SMALL LETTER THORN",
18   "LATIN SMALL LETTER Y WITH DIAERESIS", // 255
19 };
```

Y el programa siguiente muestra una tabla de todos los caracteres, cada uno con su valor numérico escrito en las tres bases habituales (decimal, octal y hexadecimal), además de su representación gráfica si es imprimible y su descripción.

latin1.cpp

```
/* Muestra el nombre oficial (en inglés) de un carácter
 * del código ISO-8859-1
 */

#include <iostream>
#include <iomanip>
#include <cctype>
#include <clocale>
#include "nombrecaracter.h"
using namespace std;

int main()
{
  if (!setlocale(LC_CTYPE, "")) cerr << "setlocale(): error" << endl;
  cout << "Dec Oct Hex Car Descripción\n"
          "--- --- --- --- -----------" << endl;
  NombreCaracter c;
  for (int i = 0; i < 256; i++) {
    c = i; // ¡CONVERSIÓN!
    cout << setw(3) << dec << i
         << setw(4) << oct << i
         << setw(4) << hex << i << ' ';
    if (isprint(i)) cout << ' ' << static_cast<char>(i) << ' ';
    else cout << "\?\?\?";
    cout << ' ' << c.nombre() << endl;
  }
}
```

La línea importante del ejemplo anterior es la asignación:

```
c = i;
```

La variable i es un *int*, mientras que c es de un tipo definido por el usuario, la clase *NombreCaracter*. El compilador no tiene información para hacer esa conversión, y el programador tiene que dársela. Como hemos definido el constructor de conversión apropiado para la clase, el compilador creará (ésta es la tarea de un constructor) un objeto invisible temporal con ese constructor, y asignará ese temporal a c, cosa que sí sabe hacer a pesar de que no hemos definido explícitamente el operador de asignación, puesto que no hace falta en este caso; pero como ya se ha dicho, el compilador efectúa una asignación miembro a miembro. O sea, la asignación anterior equivale, más explícitamente, a esto:

```
c = NombreCaracter(i);
```

Recuerde que ésta es precisamente la notación funcional del operador de modelado de C++ (§2.4.13). Por eso se llama también «operador de construcción de tipo». El compilador efectuaría algo como esto:

```
{
  NombreCaracter temp(i); // construcción del obj. temporal
  c = temp;               // asignación entre tipos iguales
}                         // se destruye el objeto temporal
```

En un ejemplo anterior considerábamos la clase *Punto*. Tenía un constructor con dos parámetros *double* con valores predeterminados. De esta manera valía por tres constructores: sin parámetros (el predeterminado), con un parámetro (el de conversión desde entero) y con dos parámetros:

```
class Punto {
 public:
  Punto(double x = 0.0, double y = 0.0): x_(x), y_(y) {}
 // ...
};
```

Según esto, podríamos convertir un *double* en un punto:

```
Punto x = 2.5;
Punto y = Punto(2.5);
Punto z(2.5);
```

Las tres formas anteriores son equivalentes; no hay asignación sino construcción. En los tres casos se crea el punto $(2.5, 0.0)$. También habría conversión en asignaciones:

```
x = -0.25;
y = Punto(-0.25);
z = static_cast<Punto>(-0.25);
```

De nuevo las tres formas son equivalentes, de menos explícita a más. Pero recuérdese que en estos casos se pasa por la creación de un objeto temporal que es el que se asigna.

Evitación de conversión en la construcción

A veces no se desea que la conversión implícita tenga lugar, y sin embargo se quiere tener un constructor de un parámetro distinto al de la clase. Para evitar la conversión entonces se declara el constructor como *explícito* poniendo ante su declaración la palabra reservada `explicit`. Si en el caso de la clase *Punto* no deseáramos que la conversión de *double* a *Punto* tuviera lugar, modificaríamos el constructor así:

```
class Punto {
 public:
  explicit Punto(double x = 0.0, double y = 0.0): x_(x), y_(y) {}
 // ...
};
```

Entonces no se podría hacer:

```
Punto x = 2.5; // ERROR, conversión prohibida
x = -0.25;     // ERROR, conversión prohibida
```

pero sí:

```
Punto y = Punto(2.5);
Punto z(2.5);
```

En el caso de y se crearía un *Punto* temporal y se construiría y con el constructor de copia.

Y en cuanto a las asignaciones, no se podría hacer:

```
x = -0.25;  // ERROR, conversión prohibida
```

pero sí:

```
y = Punto(-0.25);
z = static_cast<Punto>(-0.25);
```

Es decir, lo que se prohíbe con el constructor explícito es la conversión **implícita**, para evitar errores por despiste. Si el programador se empeña en efectuar la conversión, puede hacerla pero diciéndolo.

Asignación con conversión

En los ejemplos anteriores hemos visto que en las asignaciones se podía efectuar la conversión pero a través de la construcción de un objeto temporal. Para clases donde esto pueda dar problemas o pérdida de rendimiento, puede convenir sobrecargar el operador de asignación de forma que reciba un parámetro de un tipo distinto al de la clase. Entonces se denomina «operador de asignación con conversión» y permite efectivamente convertir del tipo del parámetro al de la clase.

Siguiendo con el ejemplo de la clase *Punto*, supongamos que tenemos otra para números complejos, *Complejo*[9], y que queremos poder convertir desde *Complejo* a *Punto*. Definiríamos un constructor de conversión en *Punto*

[9]La biblioteca estándar de C++ tiene un tipo genérico *complex* en la cabecera `<complex>`.

que recibiera un parámetro *Complejo*. Pero además, para evitar construcciones de objetos temporales, podríamos definir un operador de asignación con conversión:

```
class Complejo; // definida en otro sitio
class Punto {
  double x_, y_;
 public:
  Punto(double x = 0., double y = 0.): x_(x), y_(y) {}
  Punto(const Punto& o): x_(o.x_), y_(o.y_) {}
  Punto(const Complejo& c): x_(c.real()), y_(c.imag()) {}
  const Punto& operator =(const Punto& o)
        { x_ = o.x_; y = o.y_; return *this; }
  const Punto& operator =(const Complejo& c)
        { x_ = c.real(); y_ = c.imag(); return *this; }
  // ...
};
```

Ahora podríamos hacer cosas así (se supone que las variables cuyos nombres empiezan por *p* son *Punto*s y por *c*, *Complejo*s):

```
Punto p1 = c1;  // Ctor. de conversión Complejo->Punto
p2 = c2;        // operator =(const Complejo&)
```

La asignación anterior es directa, no se crea ningún objeto temporal ni se llama a ningún constructor.

El operador de conversión

Se acaba de ver cómo un objeto de la clase *Complejo* se puede convertir a un *Punto*, pero ¿y a la inversa? Si el autor de la clase *Punto* quiere que se produzca esta conversión, tiene que definir dentro de su clase el operador de conversión apropiado. Este operador tiene la apariencia del operador de modelado o molde de C, y es particular en cuanto que no devuelve nada, como los constructores o el destructor. Pero en este caso más bien habría que decir que no se puede escribir el tipo de devolución, porque sí que devuelve un valor, lo que pasa es que está implícito y no tenemos la opción de cambiarlo para evitar errores. De hecho debe haber una instrucción `return`. Tampoco recibe ningún parámetro aparentemente.

Ejemplo:

En la clase *Punto* añadiríamos:

```
class Punto {
 public:
  // ...
  operator Complejo()
  { // crea un objeto temporal con el constructor
    // de Complejo y lo devuelve por valor
    return Complejo(x_, y_);
  }
  // ...
};

// Ejemplo de uso
void f(Complejo& z);
Punto p;

f(p);  // Equivale a f(p.operator Complejo());
```

Hay que tener mucho cuidado y pensar bien en las conversiones que se desean. En el ejemplo anterior, ¿qué pasaría si el que diseñara la clase *Complejo* hubiera escrito en ella un constructor de conversión de *Punto* a *Complejo*?

```
class Complejo {
 public:
  Complejo(const Punto&);
  // ...
};
```

Éste le dice al compilador cómo convertir un *Punto* a un *Complejo*, ¡igual que el `operator Complejo()` en la clase *Punto*! Se produciría un error de ambigüedad al no saber el compilador qué método llamar para la conversión. Por eso, este operador debería emplearse mayormente para convertir de algún tipo fundamental a uno definido por el usuario, puesto que para los tipos fundamentales no se pueden definir operadores ni constructores. Por ejemplo, en la clase *Complejo* podría haber un constructor de conversión de *double* a *Complejo*, y un operador de conversión de *Complejo* a *double*:

```
class Complejo {
  double real, imag;
 public:
  Complejo(double re = 0.0, double im = 0.0): real(re), imag(im) {}
  void ver() const { cout << '(' << real << ", " << imag << ')'; }
  operator double() const {return sqrt(real * real + imag * imag);}
  // ...
};
```

4.4. Relaciones entre clases

Como ya se ha comentado, para construir un sistema orientado a objetos se empieza identificando los *objetos* que intervienen en él, agrupándolos en *clases* y estableciendo sus *datos* y *operaciones*. Sin embargo, en la mayoría de las ocasiones, esto sólo es el comienzo. Con estos elementos sólo se consigue describir un sistema mediante un conjunto de objetos inconexos entre sí. Aunque puede que en algunos casos esto sea suficiente, lo normal en los sistemas de mediana complejidad es que los objetos se comuniquen y colaboren entre sí intercambiando información y, por tanto, que existan *enlaces* o conexiones entre ellos.

Los enlaces particulares que relacionan a los objetos entre sí pueden abstraerse en el mundo de las clases: a cada familia de enlaces entre objetos corresponde una *relación* entre sus clases.

Al igual que los objetos son *ejemplares de las clases*, los enlaces entre objetos son *ejemplares de las relaciones* entre las clases.

Por tanto, las *relaciones* son conexiones (de diferentes tipos) que se establecen entre las clases y que vienen determinadas por las distintas formas en que las clases pueden colaborar.

Ejemplo:

En el caso de que se esté modelando un sistema de autoedición en una editorial se pueden identificar a primera vista diversos objetos, como libros, índices, capítulos, figuras y autores; se podrían agrupar en clases y así se tendría las clases *Libro*, *Índice*, *Capítulo*, *Figura* y *Autor*, que contendrían las características y comportamientos comunes de cada conjunto de objetos; y se podría guardar y solicitar información sobre cada objeto del sistema independientemente.

Pero probablemente interesaría (en un sistema de este tipo) obtener información adicional como, dado un libro, quién es su autor, o dado un autor, qué libros ha escrito. Sin embargo, con lo que se tiene construido hasta ahora en el sistema sería imposible obtenerla (los objetos son independientes unos de otros). Esta información sólo podría obtenerse si existiera de alguna forma un vínculo o conexión entre objetos del tipo *Libro* y del tipo *Autor*. Para crear esos vínculos y poder así obtener ese tipo de información es para lo que se emplean las relaciones entre clases.

Por tanto, al estudiar un sistema, no sólo hay que identificar los elementos que lo conforman, sino también cómo se relacionan entre sí.

Las relaciones entre clases se pueden clasificar en cuatro tipos especialmente importantes: *dependencias*, que representan relaciones de uso entre clases; *asociaciones* (con subtipos como las *agregaciones*), que representan relaciones estructurales entre clases; *generalizaciones/especializaciones*, que conectan clases generales con sus especializaciones, y *realizaciones*, que permiten modelar la conexión entre una interfaz y una clase.

Uno de los principios más importantes de la orientación a objetos, la *reutilización*, se consigue fundamentalmente mediante el empleo de relaciones entre clases. Por ejemplo, para describir una nueva clase no es preciso siempre partir de cero; es posible basar la definición de una clase en las de otras utilizando las relaciones de *agregación* y de *especialización*.

4.4.1. Dependencias

Una *dependencia* es una relación de uso que declara que un cambio en la especificación de una clase puede afectar a otra que la utiliza, pero no necesariamente a la inversa.

La mayoría de las veces, las dependencias se utilizarán en el contexto de las clases para indicar que una clase utiliza a otra como tipo de alguno de los parámetros de una de sus operaciones. Esto es claramente una relación de uso (si la clase utilizada cambia, la operación de la otra clase puede verse también afectada, porque la clase empleada puede presentar ahora una interfaz o comportamiento diferentes).

Ejemplo:

Supóngase que se tiene una clase *Persona* con un método para mostrar sus datos en la *salida* que se recibe como parámetro.

Como se puede ver en la figura 4.1, la *salida* es un objeto de la clase *ostream*. Por lo tanto, se establece una relación de dependencia entre la clase *Persona* y la clase *ostream*: si la clase *ostream* cambia, *mostrar()* puede verse también afectada debido a que *ostream* puede presentar ahora una interfaz o comportamiento diferentes.

4.4.2. Asociaciones

Los *enlaces* y las *asociaciones* son los medios para establecer relaciones estructurales entre objetos y clases de objetos, respectivamente. Es decir, un

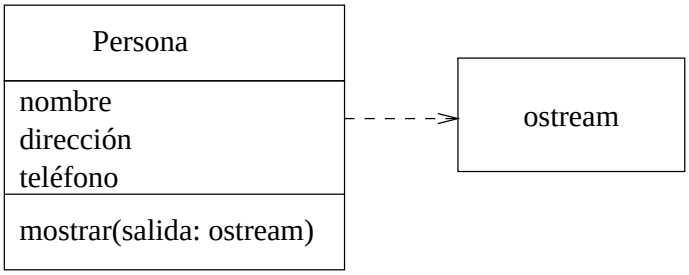

Figura 4.1: Dependencias

enlace expresa una conexión entre objetos, mientras que una *asociación* expresa una conexión entre clases.

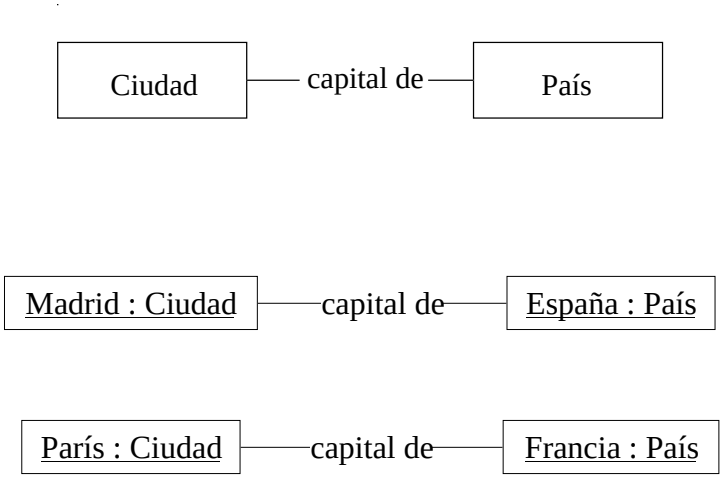

Figura 4.2: Asociaciones y enlaces

Como se ha dicho, un *enlace* sirve para representar una conexión física o conceptual entre objetos reales. Por ejemplo, *Madrid* es la capital de *España* (figura 4.2).

Una *asociación* describe un grupo de enlaces con estructura y semántica comunes. Del mismo modo que las clases son abstracciones de los objetos, las asociaciones lo son de los enlaces que existen entre los objetos. Un ejemplo de asociación entre ciudades y países es *capital de* (figura 4.2).

Las asociaciones poseen tres características principales:

Cardinalidad Indica el número de clases que intervienen en la asociación.

Multiplicidad Especifica el número de objetos que puede haber en cada

extremo de la asociación. Así, una asociación puede ser *uno a varios*, *varios a uno*, *uno a uno*, etc.

Navegabilidad Determina el sentido en el que se puede recorrer la asociación. Así, las asociaciones pueden ser *unidireccionales* o *bidireccionales*.

Las asociaciones son inherentemente bidireccionales, es decir, navegables en ambas direcciones salvo que se especifique lo contrario, a pesar de que sus nombres se suelen escribir de forma que tengan sentido tan sólo en una dirección. Es decir, dada una asociación entre dos clases, se puede navegar desde un objeto de una clase hasta un objeto de la otra, y viceversa. Por ejemplo, la asociación *capital de*, conecta de forma natural un objeto de la clase *Ciudad* con otro de la clase *País*; pero también establece una conexión en sentido contrario a la que no se da nombre, pero que podría llamarse *tiene como capital* o *su capital es*. Es un error común pensar que las asociaciones o enlaces son unidireccionales debido a que los nombres que se les dan establecen un orden natural.

Es posible que ambos extremos de una asociación estén conectados a la misma clase. Esto significa que un objeto de la clase se puede conectar con otros objetos de ella.

En la mayoría de los casos la cardinalidad de una asociación es binaria (figura 4.2), pero puede ser de orden superior, como ternaria (figura 4.3), etc. En la práctica, la inmensa mayoría va a ser binaria o calificada (una forma especial de ternaria).

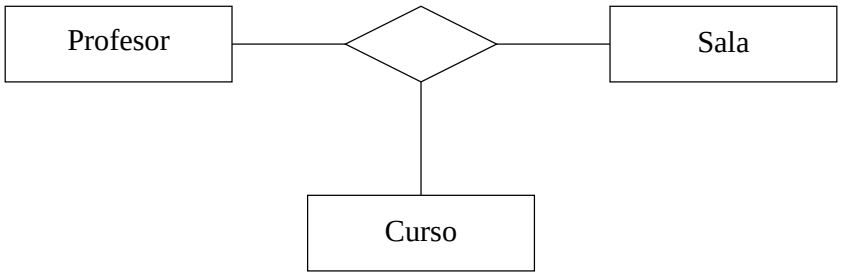

Figura 4.3: Asociación ternaria

Además, las asociaciones n-arias pueden representarse generalmente promoviendo la asociación al rango de clase y añadiendo una restricción que expresa que sus múltiples ramas se instancian todas simultáneamente, en un mismo enlace. En el ejemplo de la figura 4.4, la restricción se expresa a través de un estereotipo que indica que la clase *APSC* implementa una asociación ternaria.

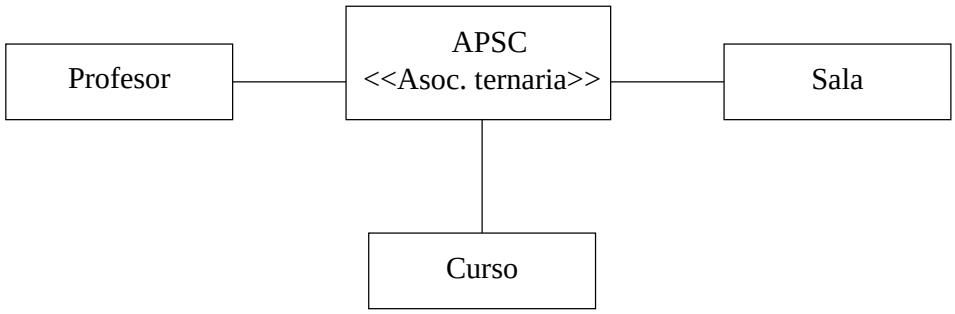

Figura 4.4: Asociación ternaria como clase estereotipada

Nombre

Una asociación puede tener un nombre, que se utiliza para describir la naturaleza de la relación (figura 4.2). Para que no haya ambigüedad en su significado, se puede dar una dirección al nombre por medio de una flecha que apunte en la dirección en la que se pretende que se lea el nombre (figuras 4.7 y 4.8).

Multiplicidad

La multiplicidad especifica cuántos objetos de una clase pueden estar relacionados con cada objeto de otra clase. Limita el número de objetos relacionados.

En el ejemplo de la figura 4.2 se ha mostrado una asociación de las que se conoce generalmente con el nombre de *uno a uno*, pues en ella se enlaza cada país con una única capital y viceversa. Se dice que su multiplicidad es 1:1.

Este caso es muy frecuente, pero no cubre todas las posibilidades. Otra forma habitual de multiplicidad es *cero o uno* y *varios*. La primera indica que en el enlace interviene tan sólo un objeto o bien ninguno, y la última expresa que el número de objetos que interviene es indeterminado, posiblemente ninguno.

En la figura 4.5 se muestra un ejemplo en el que se modela que una empresa puede emplear a un número indeterminado de personas, y que cada una de ellas trabaja para una empresa.

En cualquier caso, sólo con estas posibilidades de restricción no es completamente suficiente y se utilizan *restricciones de multiplicidad*. Éstas tan sólo se utilizan en combinación con la multiplicidad *varios* y pueden tener los tres formatos que se relacionan a continuación:

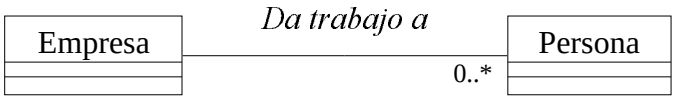

Figura 4.5: Asociación uno a varios

Cota inferior Se representa como $n+$ y se interpreta diciendo que en ese extremo del enlace deben existir n o más objetos.

Cota de conjunto Se representa como n_1, n_2, ..., n_k y se interpreta diciendo que en ese extremo deben existir n_1, n_2 , ... o n_k objetos.

Cota de rango Se representa como $m..n$ y se interpreta diciendo que en ese extremo del enlace deben existir entre m y n objetos.

Los diagramas de clases indican la multiplicidad mediante símbolos especiales al final de las líneas de asociación, como se indica en la figura 4.6.

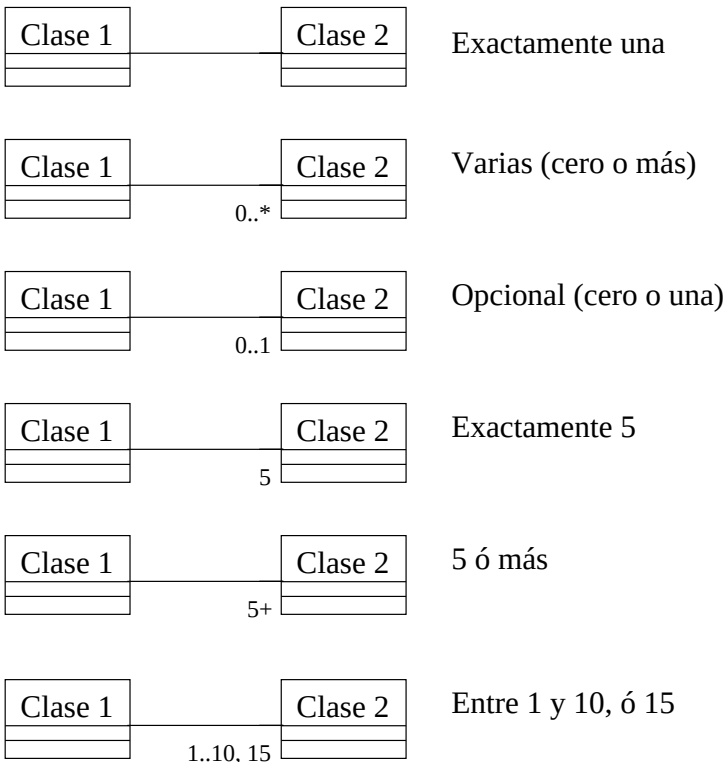

Figura 4.6: Diagramas con multiplicidades

La determinación de valores de multiplicidad óptimos es muy importante

para encontrar un equilibrio, por una parte entre flexibilidad y capacidad de extensión y por otra, entre complejidad y eficiencia.

Funciones

El extremo de una asociación se llama *función*, *papel* o *rol*. Una función da una idea del carácter con el que los objetos intervienen en un enlace, es decir, del *papel* que interpretan en él.

Toda asociación binaria posee dos funciones, una en cada extremo. La función describe cómo una clase ve a otra clase a través de la asociación.

En la figura 4.7 se presenta un ejemplo de una asociación binaria en la que se han utilizado funciones para nombrar cada uno de sus extremos.

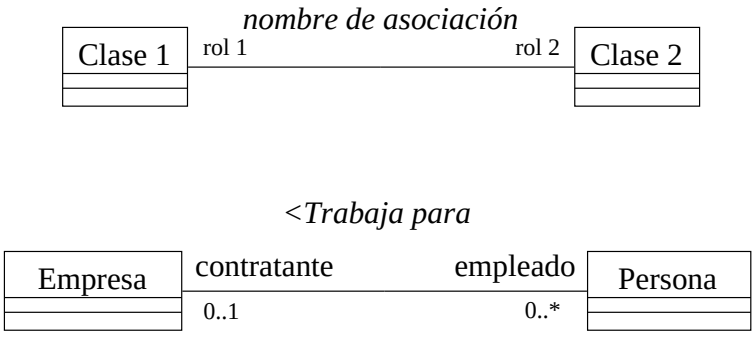

Figura 4.7: Funciones en una asociación

Las funciones son imprescindibles en las asociaciones que tienen como extremo una misma clase; también son útiles para distinguir dos asociaciones existentes entre un mismo par de clases.

Cuando solamente hay una asociación entre un par de clases distintas, los nombres de las clases suelen servir como nombres de función.

Aun cuando el nombre de la función se escribe junto a la clase de destino de una asociación, se trata en realidad de un atributo de la clase origen y debe ser único dentro de ella. Por la misma razón, ningún nombre de función debería ser igual a un nombre de atributo de la clase origen.

La presencia de un gran número de asociaciones entre dos clases puede ser sospechosa. Cada asociación aumenta el acoplamiento entre las clases asociadas y un fuerte acoplamiento puede ser el signo de una mala descomposición.

También es un error común representar varias veces la misma asociación llamando a cada asociación con el nombre de uno de los mensajes que circulan

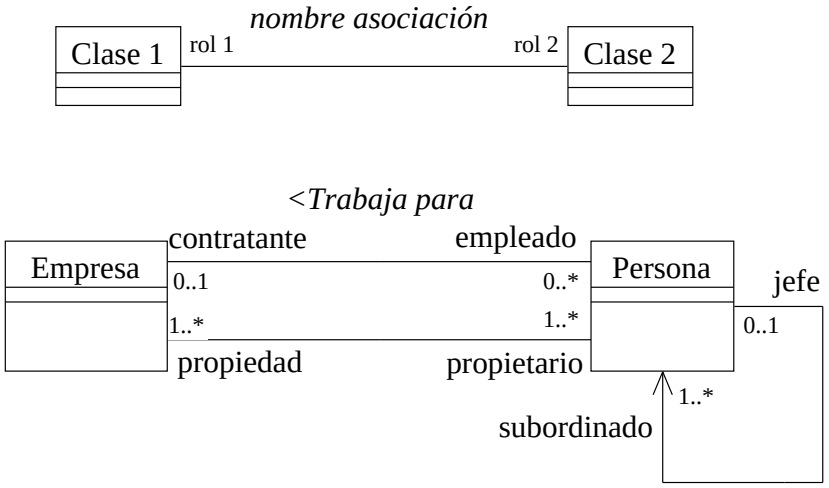

Figura 4.8: Funciones necesarias en una asociación

entre los objetos de las clases asociadas (figura 4.9).

Figura 4.9: Ejemplo de confusión entre asociación y mensajes

Agregación

Una agregación representa una asociación asimétrica en la que uno de los extremos cumple un papel predominante respecto al otro. Sea cual sea la multiplicidad, la agregación sólo afecta a una función de una asociación (figura 4.10).

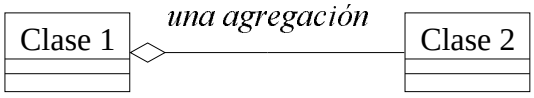

Figura 4.10: Agregación

Los siguientes criterios implican una agregación:

- Una clase forma parte de otra clase; es decir, existe una relación del tipo *todo/parte* o *está compuesto por*, donde objetos que representan componentes de algo se enlazan a otro que representa un ensamblaje completo.

  Los componentes forman parte del agregado. Las partes pueden o no existir fuera del agregado, y pueden o no aparecer en otros agregados.

- Los valores de los atributos de una clase se propagan a los de otra.

- Los objetos de una clase están subordinados a los de otra.

Las agregaciones pueden ser múltiples, como las asociaciones:

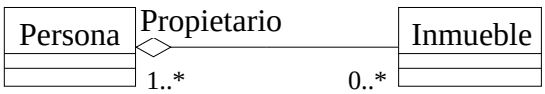

Figura 4.11: Ejemplo de agregado múltiple

La agregación es una forma especial de asociación, no un concepto independiente, aunque en ciertos casos añade connotaciones semánticas.

Si dos objetos están fuertemente acoplados mediante una relación *todo/parte*, se trata de una agregación. Si pueden considerarse independientes aunque suelan estar relacionados, entonces se trata de una asociación.

La decisión de utilizar una agregación en vez de una asociación genérica es discutible y suele ser arbitraria.

La contención física es un caso particular de agregación, llamada *composición.*

Las agregaciones pueden ser de tres tipos:

Fija Tienen una estructura fija; es decir, el número y tipo de las partes componentes están predeterminados (figura 4.12).

Variable Tienen un número fijo de niveles, pero el número de partes puede variar (figura 4.13).

Recursiva Contienen, directa o indirectamente, un ejemplar de esa misma clase de agregado; el número potencial de niveles es ilimitado.

Composición

La *composición* constituye un caso particular de agregación en la que los componentes están físicamente contenidos en el agregado.

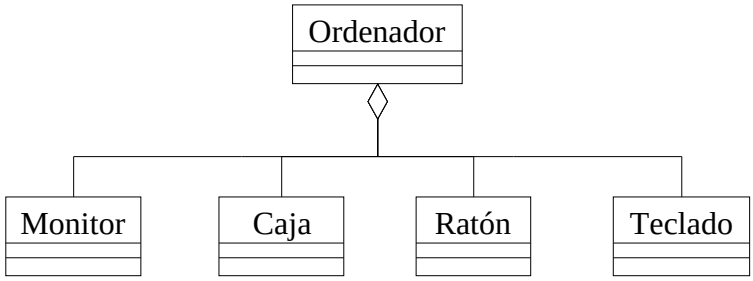

Figura 4.12: Agregación fija

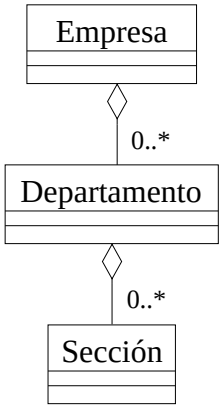

Figura 4.13: Agregación variable

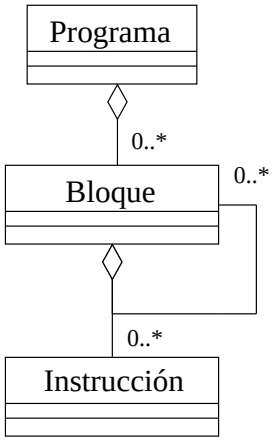

Figura 4.14: Agregación recursiva

La composición implica una restricción sobre el valor de la multiplicidad en el lado del agregado; sólo puede tomar los valores 0 ó 1 (figura 4.15).

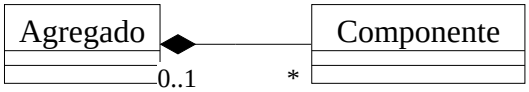

Figura 4.15: Representación gráfica de la composición

Las clases realizadas por composición se llaman también clases compuestas. Estas clases proporcionan una abstracción de sus componentes.

Agregación frente a asociación

La agregación es un concepto mucho más fuerte que el de asociación. Además, la agregación tiene diversas propiedades de las que no gozan las asociaciones.

La propiedad más interesante de la agregación es la *transitividad*, es decir, si el objeto A es parte de B y B es parte de C, entonces A también es parte de C. Por ejemplo, si la sección S forma parte del departamento D y éste de la empresa E, entonces esa sección forma parte de la empresa (figura 4.13).

En cambio, en una relación del tipo *Persona conoce a Persona*, el hecho de que *Gerardo* conozca a *Esther* y ésta a *Elvira* no implica en absoluto que *Gerardo* conozca a *Elvira*.

La agregación también se diferencia de la asociación en que posee la propiedad *antisimétrica*: si el objeto A es parte de B, entonces B no puede serlo de A.

Otra diferencia importante, desde un punto de vista semántico, es que un agregado con composición es un único objeto, aunque físicamente esté compuesto en el mundo real de objetos más sencillos que pueden tener incluso entidad independiente. En las asociaciones, en cambio, se trata con objetos individuales.

Existen casos en los que decantarse por uno u otro tipo de relación es evidente, pero en muchos otros la elección es una cuestión de juicio que se toma en cierta medida de forma arbitraria.

Navegabilidad

Las asociaciones son inherentemente bidireccionales, es decir, son navegables en ambas direcciones, a pesar de que sus nombres se suelen escribir de forma que tengan sentido tan sólo en una dirección (figura 4.16).

Por ejemplo, la asociación *trabaja para* o *empleado de*, conecta de forma natural un objeto de la clase *Persona* con otro de la clase *Empresa*; pero también establece una conexión en sentido contrario a la que no se da nombre, pero que podría llamarse *ha contratado a* o *da trabajo a* o *emplea a*.

Es un error común pensar que las asociaciones o los enlaces son unidireccionales debido a que los nombres que se les dan establecen un orden natural.

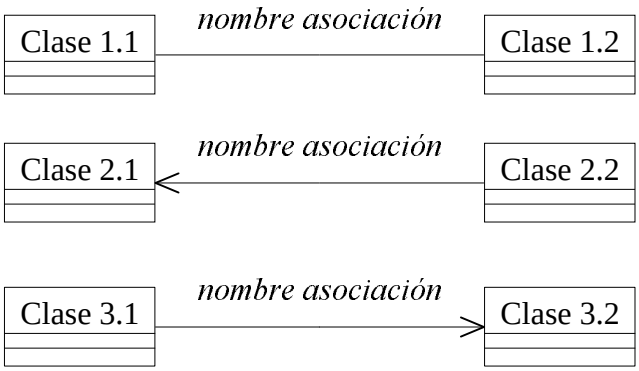

Figura 4.16: Representación de la navegabilidad de una asociación

Atributos de los enlaces

Un atributo de enlace es una propiedad de los enlaces de una asociación (figura 4.17).

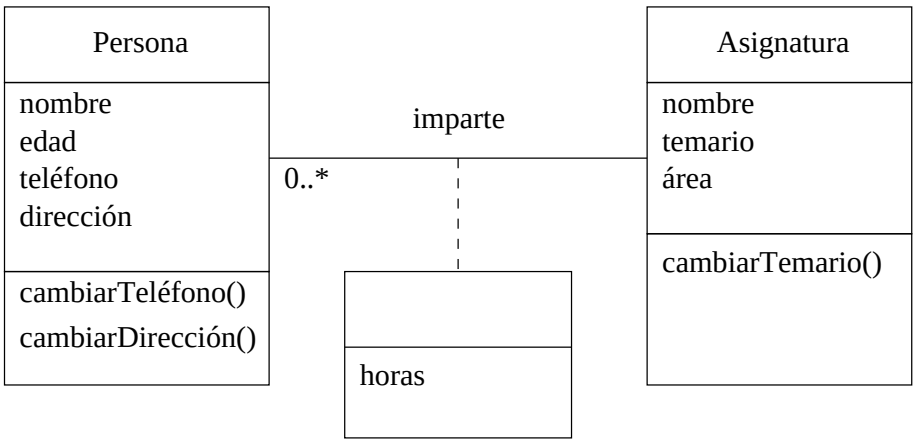

Figura 4.17: Atributos de enlace

Estos atributos son una propiedad del enlace y en general no se pueden

asociar a ninguno de los objetos individualmente sin perder información. Es posible trasladar los atributos de enlace de las asociaciones *uno a uno* y *uno a varios* a la clase que está al otro lado del *uno* (figura 4.18). Esto no es posible hacerlo con las asociaciones *varios a varios*.

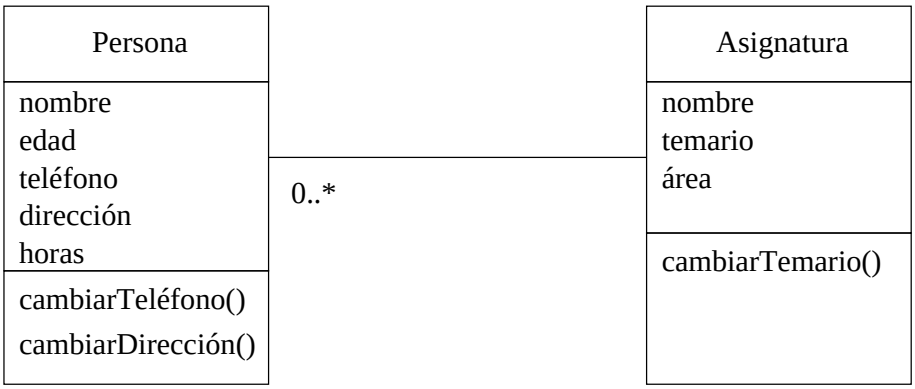

Figura 4.18: Sin atributos de enlace

Atributos de enlace frente a atributos de clase

En la figura 4.19 se muestra otro modelo para el sistema de gestión del personal de una universidad distinto del de la figura 4.18. En este modelo se tiene en cuenta que el PAS[10] de la universidad no es docente y por tanto no estaría relacionado con ninguna asignatura. Una asignatura es impartida por varias personas, y cada persona puede dar clase en una asignatura o no dar clase (en el caso del PAS).

En la figura 4.19 se ha colocado el atributo de enlace como atributo de las personas, pero este modelo no es adecuado por dos razones diferentes:

1. La *hora* no es una característica de las personas, sino de la relación que se establece entre éstas y la asignatura que imparten. Son atributos de las personas su fecha de nacimiento, su nacionalidad o el color de sus ojos, pero no el número de horas de clase que imparte de una asignatura o cuánto pagó por su automóvil (el precio es una característica del contrato de compra).

2. Considerar la hora como un atributo de las personas dificulta la facilidad para adaptar el modelo y mantener su consistencia semántica. ¿Qué ocurre con aquellas personas que no dan clase? ¿Cuál es su número de horas? Aún peor, ¿qué ocurre si posteriormente se desea añadir

[10]PAS: Personal de Administración y Servicios.

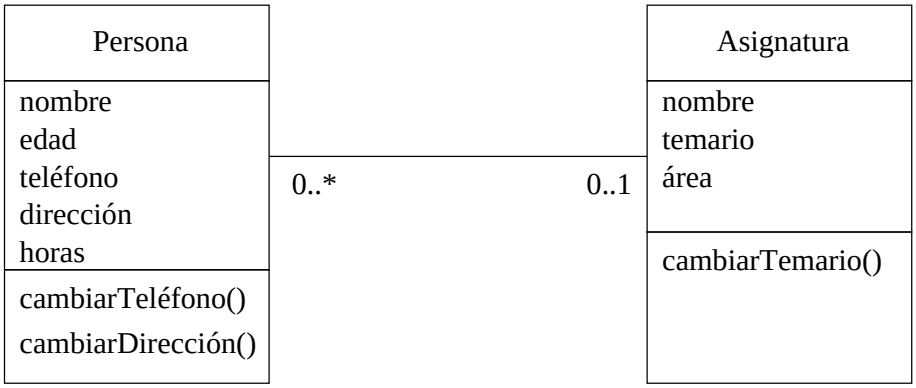

Figura 4.19: No todas las personas pueden dar clase

al modelo la posibilidad de que la misma persona imparta varias asignaturas y por lo tanto pueda impartir de cada una de ellas un número de horas distinto? En estos casos, es evidente que *hora* no puede ser un atributo de las personas, sino tan sólo de aquéllas que impartan asignaturas.

Por lo tanto la opción más adecuada es la de la figura 4.17 y, si se quiere contemplar la posibilidad de que una persona pueda impartir varias asignaturas, entonces la de la figura 4.20.

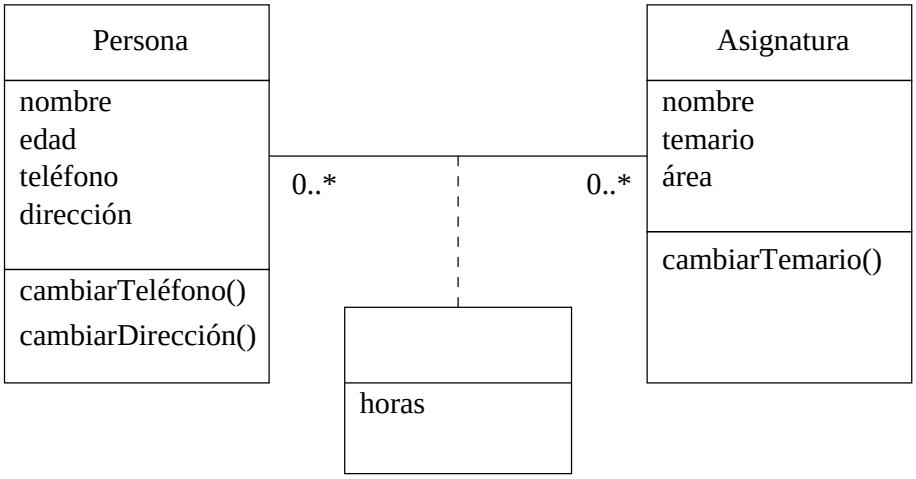

Figura 4.20: Atributos de enlace

Clases de asociación

Una asociación puede representarse mediante una clase para añadir, por ejemplo, atributos y operaciones a la asociación. Una clase de este tipo, llamada en ocasiones *clase de asociación*, es una clase como las demás y puede por ello participar en otras relaciones en el modelo (figura 4.21).

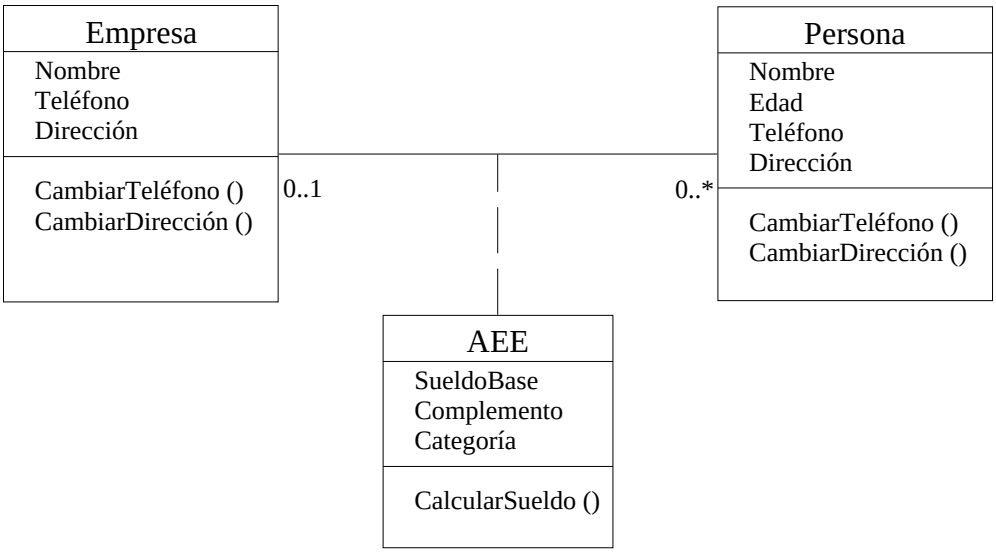

Figura 4.21: Clase de asociación

Calificación

Una asociación calificada relaciona dos clases de objetos y un calificador. Tan sólo es posible calificar las asociaciones con multiplicidad *varios* en uno de sus extremos, y el calificador permite distinguir objetos individuales en ese extremo. El calificador es un atributo especial que reduce la multiplicidad efectiva de una asociación.

Las asociaciones *uno a varios* y *varios a varios* pueden ser calificadas. El calificador distingue entre el conjunto de objetos que se encuentra en el extremo *varios* de la asociación (figura 4.22). Una asociación calificada también se puede considerar como una forma de asociación ternaria.

4.4.3. Generalizaciones y especializaciones

La generalización es la relación que existe entre una clase y una o más versiones especializadas de esa misma clase, es decir, la relación de clasificación

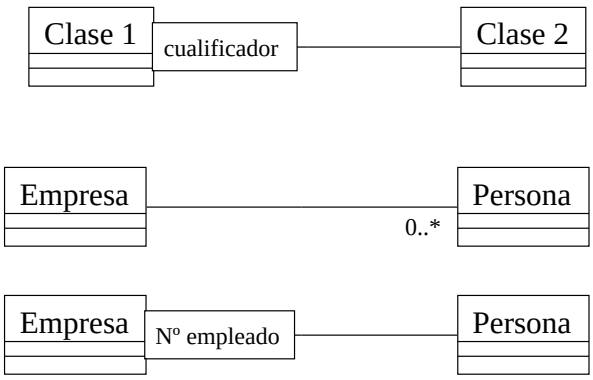

Figura 4.22: Asociación calificada

entre un elemento más general y otro más específico. La que se está especializando se denomina *superclase* y cada versión especializada se conoce como *subclase*.

La *subclase*, también llamada clase *hija* o *derivada*, hereda la descripción de la *superclase*, también conocida como clase *madre* o *base*. Por tanto, los atributos y operaciones comunes a un grupo de subclases se asocian a la superclase y son compartidos por todas las subclases.

La generalización y la especialización son dos puntos de vista distintos de la misma relación vista desde la superclase o desde las subclases. La subclase especializa la superclase, la superclase generaliza las subclases.

Una clase derivada puede ser a su vez base de otra; así se puede construir una jerarquía de tipos. Una clase puede no tener base o, tener uno o más bases. Una clase sin bases se denomina *clase raíz*. Una clase sin derivadas se llama *clase hoja*.

Un objeto de una clase lo es a su vez de todas sus antecesoras. Por tanto, todas las características de las clases antecesoras deben ser aplicables a los objetos de las subclases. Una clase descendiente no puede omitir o suprimir un atributo de un antecesor, ya que de esta forma no sería un verdadero objeto de dicho antecesor. Exactamente lo mismo ocurre con las operaciones. Una subclase puede volver a implementar una operación, sin modificar su signatura. Esto se llama *redefinición*.

No obstante, las subclases pueden añadir nuevas características. Esto se denomina *extensión*.

La generalización se denomina a veces «relación *es una*» porque todo objeto de una subclase *es un* objeto de la superclase al mismo tiempo.

La generalización y la especialización son relaciones transitivas.

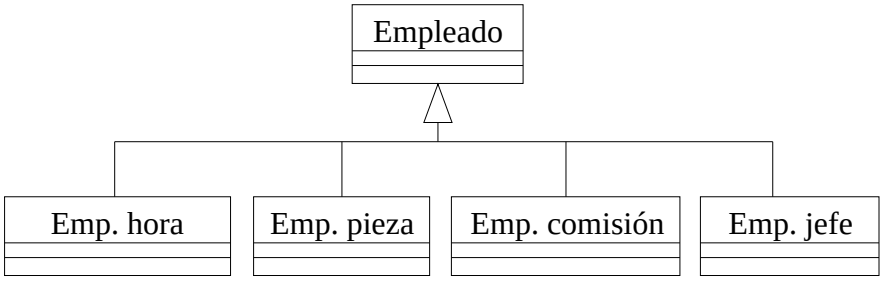

Figura 4.23: Ejemplo de generalización

La generalización facilita el modelado estructurando las clases, capturando de forma concisa lo que es similar y lo que es distinto con respecto a las clases.

Los lenguajes orientados a objetos proporcionan apoyo para la creación de relaciones de generalización/especialización a través del mecanismo de la *herencia*.

Generalización, especialización y herencia son tres conceptos muy relacionados que constituyen la herramienta de abstracción más potente que suministra la orientación a objetos, pues permiten generalizar los conceptos que representan los objetos abstrayendo de ellos sus características de más alto nivel. También ofrecen la posibilidad de clasificar y agrupar clases que tienen características similares.

Se utilizará el término «generalización» para hacer alusión a la relación entre clases, mientras que «herencia» aludirá al mecanismo empleado por el lenguaje para implementar estas relaciones.

La generalización es un potente mecanismo de reutilización de código que sirve para definir una clase en función de otra, de la cual *deriva*.

Tipos de herencia

- Herencia simple

  Cada subclase sólo tiene una superclase. Un ejemplo se puede ver en la figura 4.23.

- Herencia múltiple

  La herencia múltiple permite que una clase tenga más de una superclase, y que herede características de todas sus antecesoras.

Proporciona mayor potencia para especificar clases y mayor oportunidad de reutilización de código. Pero en cambio, se pierde sencillez conceptual y de implementación.

Una clase con más de una superclase se denomina *clase unión* y hereda entre sus atributos todos los definidos para esas superclases (figuras 4.24 y 4.25).

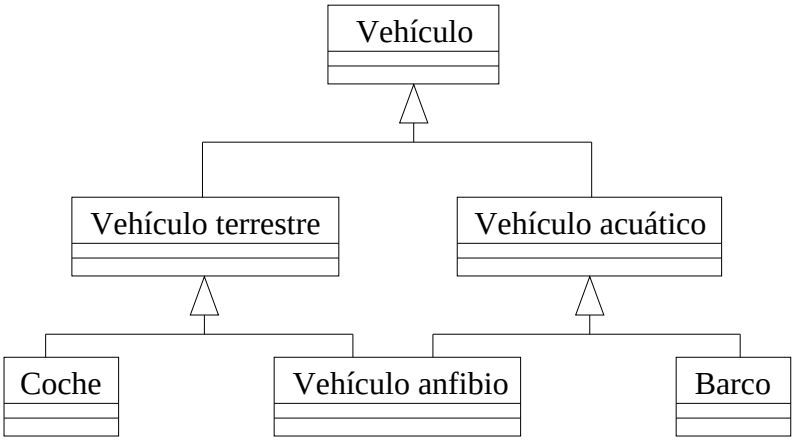

Figura 4.24: Herencia múltiple

Las clases pueden tener herencia múltiple procedente de distintas generalizaciones, o de diferentes clases dentro de una generalización con solapamiento (clases hermanas), pero nunca de dos clases procedentes de una misma generalización disjunta.

Una clase puede ser especializada según varios criterios simultáneamente. Cada criterio de la generalización se indica en el diagrama asociando un discriminante a la relación de generalización.

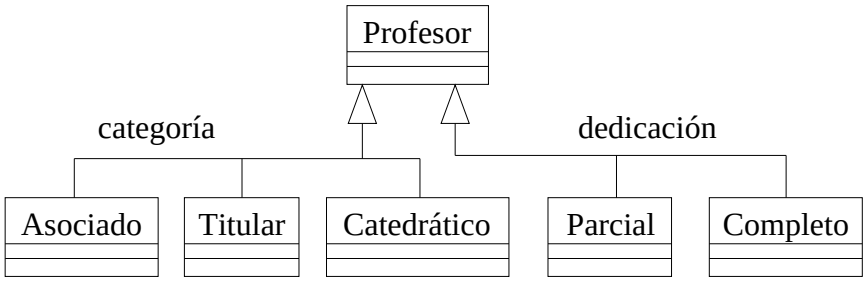

Figura 4.25: Ejemplo de generalización según criterios independientes

Asociación frente a generalización

No hay que confundir la asociación con la generalización. La asociación es una relación entre clases donde están implicados, al menos, dos objetos diferentes. La generalización es una relación entre clases, es una forma de refinar o estructurar la descripción de un único objeto.

Mediante la generalización, un objeto es a la vez un ejemplar de la superclase y uno de la subclase.

En algunos casos puede ser complicado distinguir entre generalización y asociación. Considerese por ejemplo un editor de textos que se ejecuta en una ventana. ¿Cuál de los dos modelos recogidos en la figura 4.26 es el más adecuado?

En la figura 4.26(a) se ha modelado el problema diciendo que cada objeto de la clase *Editor se ejecuta en* un objeto de la clase *Ventana* y que cada una de éstas puede ejecutar un editor. En cambio, en la figura 4.26(b), *Editor* hereda sus propiedades de la clase *Ventana*, por lo que un editor se podría considerar como una *ventana editable*.

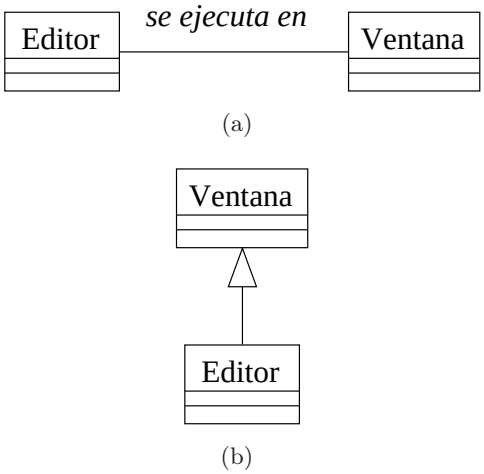

Figura 4.26: ¿Heredar o asociar?

Para responder al dilema de qué modelo elegir, suele ser conveniente plantearse las siguientes preguntas: ¿es un editor una ventana?, ¿es lógico pensar en que un editor es una especialización de ventana? O lo que es lo mismo: ¿es posible usar un editor en todas aquellas situaciones en las que se emplea una ventana?

Parece que la respuesta a ambas preguntas es no: un editor no es una ventana, tan sólo se ejecuta dentro de ella, y un editor tampoco es un tipo especial de

ventana. Por lo tanto, en este caso, parece que el modelo más lógico es el de la figura 4.26(a). Si aún no está convencido, trate de modelar un editor que se ejecute simultáneamente en varias ventanas, cada una de las cuales ofrezca una vista diferente del mismo texto. ¿Cuál de los dos modelos recogería mejor esta situación?

Las dos preguntas que se han planteado en el párrafo anterior suelen ofrecer un buen criterio para determinar qué tipo de estructura utilizar.

Polimorfismo

La forma más pura de polimorfismo en la orientación a objetos se consigue redefiniendo una operación a lo largo de una misma jerarquía de clases.

Una operación es polimórfica cuando se define con la misma signatura en diferentes puntos de una jerarquía de clases, de modo que una clase redefine el comportamiento que tenía en una clase antecesora.

Cuando se envía un mensaje sobre una referencia de objeto la operación de la jerarquía que se ejecuta se elige, en general, polimórficamente. Esto implica que el tipo del objeto receptor en tiempo de ejecución determina la elección de la operación concreta a ejecutar.

Clases abstractas

Una *clase abstracta* es aquélla que posee subclases, pero de la que no pueden existir objetos por ser abstracta alguna de sus operaciones. Una *operación abstracta* es aquélla de la que sólo se especifica su signatura y no su definición.

Cada subclase no abstracta de una clase abstracta deberá proporcionar una definición para cada operación abstracta heredada.

Una clase abstracta representa la máxima abstracción posible dentro de un sistema, es decir, recoge la esencia de un concepto que se repite en muchas ocasiones con ligeras variantes. Por lo tanto, las clases abstractas proporcionan un mecanismo para favorecer la reutilización.

En el ejemplo de la figura 4.23, *Empleado* podría ser una clase abstracta que sirviera para recoger las características generales de todos los empleados, sin ligadura alguna con ningún tipo particular de empleado. Por lo tanto, sería fácil reutilizar esta clase en cualquier otro proyecto dentro de un dominio similar.

4.4.4. Realizaciones

Una realización es una relación que permite modelar la conexión entre una *interfaz* y una clase.

La realización es lo suficientemente diferente de la dependencia, la generalización y la asociación como para ser tratada como un tipo aparte de relación. Semánticamente, la realización es algo así como una mezcla entre dependencia y generalización.

Interfaces

Una interfaz representa una colección de operaciones que sirven para especificar un servicio de una clase.

Lo más interesante de las interfaces es que permiten separar la especificación de su implementación. Esto es, al declarar una interfaz, se puede enunciar el comportamiento deseado de una abstracción independientemente de su implementación. Los clientes pueden trabajar con esa interfaz, y se puede construir cualquier implementación de ella, siempre que ésta satisfaga las responsabilidades indicadas en la interfaz.

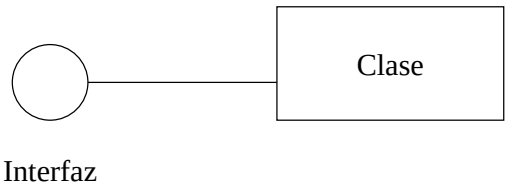

Figura 4.27: Interfaces

Al contrario que las clases, las interfaces no especifican ninguna estructura (así que no pueden incluir atributos) ni ninguna implementación (así que no pueden incluir métodos, que proporcionan la implementación de una operación). Como una clase, una interfaz puede incluir cualquier número de operaciones.

Al igual que las clases, las interfaces pueden participar en relaciones de generalización, asociación, dependencia y realización.

Una interfaz puede ser realizada por muchas clases, y una clase puede realizar muchas interfaces. Al hacer esto, la clase se compromete a cumplir fielmente las especificaciones, lo que significa que proporcionará un conjunto de métodos que implementen apropiadamente las operaciones definidas en la interfaz. Análogamente, una clase puede depender de varias interfaces.

Las interfaces son similares a las clases abstractas, ninguna de las dos puede tener instancias directas. En cambio, una clase abstracta puede tener atributos y también implementar algunas operaciones, pero una interfaz no.

4.5. Diseño de las relaciones entre clases

Las relaciones proporcionan las vías de acceso entre los objetos. Durante el diseño hay que formular una estrategia de implementación.

Independientemente de la estrategia seleccionada, hay que ocultar la implementación de las relaciones empleando operaciones de acceso para recorrerlas y actualizarlas.

4.5.1. Diseño de las dependencias

No hay que tomar ninguna decisión de diseño al implementar las relaciones de dependencia, ya que están implícitas en el lenguaje.

Las dependencias deben reducirse en todo lo posible, ya que aumentan el acoplamiento entre clases. Para facilitar su manejo suele emplearse compilación por separado de las clases dependientes.

4.5.2. Diseño de las asociaciones

La forma más habitual de diseñar las asociaciones suele ser añadiendo atributos adicionales en las clases que intervienen en dicha relación. Esos atributos hacen referencia explícita a otros objetos.

Por ejemplo, para implementar la asociación *capital de*, a la clase *Ciudad* se le añadiría un atributo, llámese *país*, que haría referencia a un objeto del tipo *País*, y a la clase *País* se le añadirá un atributo, llámese *capital*, que haría referencia a un objeto del tipo *Ciudad*.

De esta forma, dado un país se puede saber cuál es su capital accediendo a su atributo *capital*. Por otro lado, dada una ciudad (capital de algun país) se puede saber cuál es el país del que es capital accediendo a su atributo *país*.

El diseño de asociaciones de esta forma es perfectamente admisible y de lo más común, aunque lo ideal sería que las asociaciones no se modelaran de esta manera, sino como clases de asociación aparte.

Asociaciones monodireccionales

Las asociaciones monodireccionales son las que sólo se van a recorrer en un sentido, es decir, sólo van a ser navegables en una dirección.

Siguiendo con el ejemplo de la relación *capital de*, supóngase que en el sistema lo único que se quiere saber es, dado un país, cuál es su capital, no preguntándose nunca de qué país es capital una determinada ciudad. En este caso, no es necesario implementar la relación en las dos direcciones porque nunca se va a acceder de *Ciudad* a *País*.

Por tanto, el diseño de este tipo de asociaciones se puede simplificar. Es posible implementarlas mediante un único atributo en el sentido que interese.

En nuestro ejemplo, sólo sería necesario implementar la asociación de *País* a *Ciudad*, es decir, añadir un atributo en la clase *País* que haría referencia a un objeto del tipo *Ciudad*, evitándose así tener que añadir un atributo a la clase *Ciudad*.

Por otro lado, en el diseño de las asociaciones no hay que olvidar la multiplicidad. Dependiendo de cuál sea se actuará de una manera u otra:

- Si la multiplicidad es de *uno*, el atributo hará referencia a un único objeto.
- Si la multiplicidad es *varios*, el atributo será un conjunto de referencias a objeto. Si el extremo *varios* está ordenado se puede utilizar una secuencia en lugar de un conjunto.

En cuanto a las asociaciones calificadas:

- Si tienen multiplicidad *uno*, se pueden implementar en forma de *diccionario* (un diccionario es una aplicación entre un conjunto de claves y un conjunto de valores).
- Si tienen multiplicidad *varios*, se pueden implementar como un *diccionario multivalor* (a cada clave se puede asociar más de un valor).

Asociaciones bidireccionales

Se pueden implementar de varias formas:

- Como una asociación monodireccional, realizándose una búsqueda cuando se requiere recorrer la asociación en sentido contrario.

Esto sólo se debe utilizar cuando se realizan pocos accesos en sentido contrario y se desean minimizar los costes de modificación y almacenamiento.

- Como dos asociaciones monodireccionales, una en cada sentido.

  Esto permite un acceso rápido, pero si se modifica alguno de los enlaces de una de las asociaciones, también su inverso deberá modificarse para mantener la congruencia. Este enfoque es útil cuando el número de accesos supera al de modificaciones.

- Como una clase de asociación separada, independiente de ambas, de manera que un objeto de la clase de asociación contendrá un conjunto de parejas de referencias a objeto. Una clase de asociación se puede implementar eficientemente empleando dos diccionarios, uno para cada sentido.

  Esto es útil para relacionar clases ya existentes que no se puedan modificar, por ejemplo las de una biblioteca de clases, porque la clase de asociación se puede crear sin necesidad de agregar ningún atributo a las clases relacionadas.

Atributos de enlace

Si una asociación tiene atributos de enlace, su diseño depende de la multiplicidad.

- Si la asociación es *uno a uno*, los atributos de enlace se pueden almacenar como atributos de cualquiera de las clases.
- Si la asociación es *varios a uno*, los atributos de enlace se pueden almacenar como atributos de la clase *varios*, por cuanto cada objeto del lado *varios* aparece una sola vez en la asociación.
- Si la asociación es *varios a varios*, no se pueden asociar los atributos de enlace con una u otra clase; la mejor implementación suele ser como una clase de asociación aparte, en la que cada objeto represente un enlace y sus atributos.

Agregaciones

Para el diseño de las agregaciones se sigue el mismo esquema que con las asociaciones con algunas diferencias:

- Normalmente son monodireccionales.

- En el caso de que la existencia de los objetos (componentes) que forman parte de otro (agregado) no tenga sentido independientemente de la existencia de este último (**composición**), entonces, en lugar de utilizar atributos que hagan referencia a los objetos, se pueden utilizar los propios objetos.

Considérese una agregación entre *Libro* y *Capítulo*. En este caso se podría pensar que la existencia de los *capítulos* no tiene sentido independientemente de que exista un objeto de la clase *Libro*, por lo que para implementar la agregación se añadiría un atributo en la clase *Libro* que sería una secuencia de objetos *Capítulo*.

4.5.3. Diseño de las generalizaciones

Prácticamente no hay que tomar ninguna decisión de diseño al implementar generalizaciones, ya que casi todos (por no decir todos) los lenguajes orientados a objetos proporcionan estructuras sencillas y fáciles de utilizar para llevar esto a cabo.

El mayor problema que se puede presentar es que, habiendo empleado herencia múltiple en el modelado, nuestro lenguaje no la permita[11]. En tal caso, existen técnicas que permiten reestructurarla en herencia simple.

También hay que decir que a medida que progresa el diseño de objetos es frecuente que se puedan ajustar las definiciones de las clases y de las operaciones para incrementar el empleo de la herencia.

El diseñador debería:

- Reorganizar y ajustar las clases y operaciones para incrementar la herencia.

  Puede ocurrir que una misma operación se defina a través de varias clases y es fácil que sea heredada; sin embargo es muy frecuente que las operaciones aplicadas a diferentes clases sean parecidas pero no idénticas, y con pequeñas modificaciones se puedan hacer coincidir las definiciones de dichas operaciones de forma que se pueda emplear la herencia.

  Es frecuente añadir nuevas clases y operaciones durante el diseño.

  Cuando se ha reconocido un comportamiento común, se puede crear una superclase que implemente las características compartidas, dejando solamente las especializadas en las subclases. Esta transformación del

[11] Como se verá, C++ sí posee herencia múltiple.

modelo de objetos se conoce como *extraer una superclase*. Es normal que la superclase resultante sea abstracta.

Ventajas de las clases abstractas:

- Favorece la reutilización.
- Aumenta la modularidad.
  La descomposición de una clase en dos clases, una concreta y otra abstracta, que separen los aspectos específicos y los generales de la clase, respectivamente, ayudan a conseguir una mayor modularidad.
- Facilita el mantenimiento.

- Emplear la delegación para compartir la implementación.

  Con la herencia, el comportamiento de una superclase es compartido por todas sus subclases. Esto es factible cuando se puede decir que la subclase es una forma de la superclase.

  En algunas ocasiones, los programadores utilizan la herencia como una técnica de implementación, sin la intención de garantizar este comportamiento. Con frecuencia, una clase ya existente implementa parte del comportamiento que se desea ofrecer en una clase de nueva definición, aunque en otros aspectos las dos clases sean diferentes.

  Muchas veces se pretende heredar de la clase existente para lograr una parte de la implementación de la nueva clase. Esto puede dar lugar a problemas si las demás operaciones que se heredan proporcionan un comportamiento no deseado.

  Ejemplo:

  Supóngase que se quiere implementar una clase *Pila* y se dispone ya de una clase *Lista*. Ya que la clase *Lista* contiene todo lo que se necesita, se podría pensar en construir la clase *Pila* heredando de ella. Para apilar un elemento se añade al principio de la lista y para desapilar se elimina el primer elemento de la lista.

  Pero al heredar, también se están introduciendo en la clase *Pila* operaciones propias de las listas que no tiene sentido realizar sobre las pilas. Si estas operaciones no deseadas se utilizan sobre una pila, ésta no se comportará como tal.

  Para evitar este problema en la reutilización de la clase *Lista*, se puede definir la clase *Pila* como un agregado (en realidad, una composición) de la otra clase. De esta manera se pueden definir sólo las operaciones propias de la clase *Pila* delegando selectivamente su comportamiento en las de la clase *Lista*.

La delegación consiste en desviar una operación aplicable a un objeto a otro objeto que forma parte del primero o que está relacionado con él. Es decir, la delegación es un mecanismo de implementación por el que un objeto traspasa la responsabilidad de una operación a otro para que sea éste quien la ejecute. Sólo se delegan las operaciones requeridas de modo que no hay peligro de heredar «por accidente» operaciones que carezcan de sentido.

4.5.4. Diseño de las realizaciones

En algunos lenguajes orientados a objetos no hay que tomar ninguna decisión de diseño para las realizaciones, ya que proporcionan estructuras sencillas y fáciles de utilizar para llevar esto a cabo.

En el caso de que el lenguaje no las incorpore[12] se pueden diseñar utilizando clases abstractas como interfaces y especializaciones de ellas como implementaciones.

4.6. Implementación de las relaciones en C++

4.6.1. Implementación de las asociaciones

A grandes rasgos existen dos maneras de implementarlas: mediante atributos y métodos en las clases implicadas o mediante clases de asociación explícitas (la mayoría de los lenguajes, y C++ no es una excepción, no las incorporan).

En C++ la biblioteca estándar de plantillas (STL) proporciona las clases contenedoras necesarias para implementar las asociaciones de manera eficiente y sencilla; por ejemplo, listas (*list*), conjuntos (*set*), multiconjuntos (*multiset*) y aplicaciones (*map* y *multimap*).

Atributos y métodos en las clases implicadas

Todos los atributos y métodos necesarios para implementar las asociaciones se añaden como miembros a las definiciones de las clases implicadas en la asociación.

Una asociación binaria se implementa, normalmente, con dos miembros de datos, uno para cada una de las clases asociadas.

En el caso de que un extremo de la asociación sea de multiplicidad *uno* el

[12] Como ocurre en C++.

miembro de datos que le corresponde será un puntero a un objeto de la otra clase.

Ejemplo:

Véase cómo se implementaría una asociación *uno a uno* entre la clase *Persona* y la clase *Asignatura*.

uno-uno/persona.h

```
1  #ifndef PERSONA_H_
2  #define PERSONA_H_
3
4  #include <string>
5  using std::string;
6  class Asignatura;               // declaración adelantada
7
8  class Persona {
9  public:
10   Persona(string nombre, /* ... */ string direccion);
11   // ...
12   void mostrar() const;
13   void imparte(Asignatura& asignatura);
14   void mostrarAsignatura() const;
15 private:
16   string nombre;
17   // ...
18   string direccion;
19   Asignatura* asignatura;
20 };
21
22 #endif
```

uno-uno/persona.cpp

```
1  #include "persona.h"
2  #include "asignatura.h"
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  // Constructor
8
9  Persona::Persona(string nombre, /* ..., */ string direccion):
10   nombre(nombre), /* ..., */ direccion(direccion),
11   asignatura(0)
12 {}
```

```
13
14 // Muestra los datos de una persona
15
16 void Persona::mostrar() const
17 {
18   cout << "Nombre:     " << nombre << "\n"
19        // ...
20        << "Dirección:  " << direccion << endl;
21 }
22
23 // Asociación: una persona imparte una asignatura
24
25 void Persona::imparte(Asignatura& asignatura)
26 {
27   this->asignatura = &asignatura;
28 }
29
30 // Muestra la asignatura impartida por una persona
31
32 void Persona::mostrarAsignatura() const
33 {
34   if (!asignatura)
35     cout << "No imparte ninguna asignatura" << endl;
36   else
37     asignatura->mostrar();
38 }
```

uno-uno/asignatura.h

```
1  #ifndef ASIGNATURA_H_
2  #define ASIGNATURA_H_
3
4  #include <string>
5  using std::string;
6  class Persona;                  // declaración adelantada
7
8  class Asignatura {
9  public:
10   Asignatura(string nombre, /* ..., */ string area);
11   // ...
12   void mostrar() const;
13   void impartida(Persona& persona);
14   void mostrarPersona() const;
15 private:
16   string nombre;
17   // ...
18   string area;
19   Persona* persona;
20 };
```

```
21
22 #endif
```

uno-uno/asignatura.cpp

```
1  #include "asignatura.h"
2  #include "persona.h"
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  // Constructor
8
9  Asignatura::Asignatura(string nombre, /* ..., */ string area):
10   nombre(nombre), /* ..., */ area(area),
11   persona(0)
12 {}
13
14 // Muestra los datos de una asignatura
15
16 void Asignatura::mostrar() const
17 {
18   cout << "Asignatura: " << nombre << "\n"
19        // ...
20        << "Área:       " << area << endl;
21 }
22
23 // Asociación: una asignatura es impartida por una persona
24
25 void Asignatura::impartida(Persona& persona)
26 {
27   this->persona = &persona;
28 }
29
30 // Muestra la persona que imparte una asignatura
31
32 void Asignatura::mostrarPersona() const
33 {
34   if (!persona)
35     cout << "No es impartida por ninguna persona" << endl;
36   else
37     persona->mostrar();
38 }
```

uno-uno/prueba.cpp

```
1  #include "persona.h"
2  #include "asignatura.h"
```

```
3
4  int main()
5  {
6    Persona genaro("Genaro López Sánchez",
7                   // ...
8                   "C/ Chile s/n");
9    Asignatura mtp("Metodología y Tecnología de la Prog.",
10                  // ...
11                  "Lenguajes y Sistemas Informáticos");
12
13   genaro.mostrar();
14   genaro.mostrarAsignatura();
15   // Genaro imparte MTP
16   genaro.imparte(mtp);
17   genaro.mostrar();
18   genaro.mostrarAsignatura();
19   mtp.mostrar();
20   mtp.mostrarPersona();
21   // MTP es impartida por Genaro
22   mtp.impartida(genaro);
23   mtp.mostrar();
24   mtp.mostrarPersona();
25 }
```

En lugar de punteros se podrían utilizar referencias, simplificándose ligeramente la manipulación, pero obligando así a que el enlace se realice durante la creación del objeto origen.

Si un extremo de la asociación es de multiplicidad *varios* el miembro de datos que le corresponde será un conjunto de punteros a objetos de la otra clase.

Ejemplo:

Véase ahora cómo sería la implementación de una asociación *varios a varios* entre las clases del ejemplo anterior.

La implementación emplea la clase paramétrica *set* de la STL, que se verá en detalle en el capítulo 7.

varios-varios/persona.h

```
1  #ifndef PERSONA_H_
2  #define PERSONA_H_
3
4  #include <string>
5  #include <set>
```

```
6  using std::string;
7  using std::set;
8  class Asignatura;               // declaración adelantada
9
10 class Persona {
11 public:
12   Persona(string nombre, /* ... */ string direccion);
13   // ...
14   void mostrar() const;
15   void imparte(Asignatura& asignatura);
16   void mostrarAsignaturas() const;
17 private:
18   string nombre;
19   // ...
20   string direccion;
21   typedef set<Asignatura*> Asignaturas;
22   Asignaturas asignaturas;
23 };
24
25 #endif
```

varios-varios/persona.cpp

```
1  #include "persona.h"
2  #include "asignatura.h"
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  // Constructor
8
9  Persona::Persona(string nombre, /* ..., */ string direccion):
10   nombre(nombre), /* ..., */ direccion(direccion)
11 {}
12
13 // Muestra los datos de una persona
14
15 void Persona::mostrar() const
16 {
17   cout << "Nombre:     " << nombre << "\n"
18        // ...
19        << "Dirección:  " << direccion << endl;
20 }
21
22 // Asociación: una persona imparte una asignatura
23
24 void Persona::imparte(Asignatura& asignatura)
25 {
26   asignaturas.insert(&asignatura);
```

```
27 }
28
29 // Muestra las asignaturas impartida por una persona
30
31 void Persona::mostrarAsignaturas() const
32 {
33   if (asignaturas.empty())
34     cout << "No imparte ninguna asignatura" << endl;
35   else
36     for (Asignaturas::iterator i = asignaturas.begin();
37          i != asignaturas.end(); ++i)
38       (*i)->mostrar();
39 }
```

varios-varios/asignatura.h

```
1  #ifndef ASIGNATURA_H_
2  #define ASIGNATURA_H_
3
4  #include <string>
5  #include <set>
6  using std::string;
7  using std::set;
8  class Persona;                  // declaración adelantada
9
10 class Asignatura {
11 public:
12   Asignatura(string nombre, /* ..., */ string area);
13   // ...
14   void mostrar() const;
15   void impartida(Persona& persona);
16   void mostrarPersonas() const;
17 private:
18   string nombre;
19   // ...
20   string area;
21   typedef set<Persona*> Personas;
22   Personas personas;
23 };
24
25 #endif
```

varios-varios/asignatura.cpp

```
1  #include "asignatura.h"
2  #include "persona.h"
3  #include <iostream>
4  #include <string>
```

```
5  using namespace std;
6
7  // Constructor
8
9  Asignatura::Asignatura(string nombre, /* ..., */ string area):
10   nombre(nombre), /* ..., */ area(area)
11 {}
12
13 // Muestra los datos de una asignatura
14
15 void Asignatura::mostrar() const
16 {
17   cout << "Asignatura: " << nombre << "\n"
18        // ...
19        << "Área:       " << area << endl;
20 }
21
22 // Asociación: una asignatura es impartida por varias personas
23
24 void Asignatura::impartida(Persona& persona)
25 {
26   personas.insert(&persona);
27 }
28
29 // Muestra las personas que imparten una asignatura
30
31 void Asignatura::mostrarPersonas() const
32 {
33   if (personas.empty())
34     cout << "No es impartida por ninguna persona" << endl;
35   else
36     for (Personas::iterator i = personas.begin();
37          i != personas.end(); ++i)
38       (*i)->mostrar();
39 }
```

varios-varios/prueba.cpp

```
1  #include <iostream>
2  #include "persona.h"
3  #include "asignatura.h"
4  using namespace std;
5
6  int main()
7  {
8    Persona genaro("Genaro López Sánchez",
9                   // ...
10                  "C/ Chile s/n");
11   Persona marisa("Marisa Gómez Jiménez",
```

```
12                  // ...
13                  "C/ Argentina s/n");
14    Persona felisa("Felisa González Pérez",
15                  // ...
16                  "C/ Bolivia s/n");
17    Asignatura mtp("Metodología y Tecnología de la Prog.",
18                  // ...
19                  "Lenguajes y Sistemas Informáticos");
20    Asignatura ada("Análisis y Diseño de Algoritmos",
21                  // ...
22                  "Lenguajes y Sistemas Informáticos");
23    Asignatura poo("Programación Orientada a Objetos",
24                  // ...
25                  "Lenguajes y Sistemas Informáticos");
26
27    // Genaro imparte MTP
28    genaro.imparte(mtp); mtp.impartida(genaro);
29    // Marisa imparte MTP y ADA
30    marisa.imparte(mtp); mtp.impartida(marisa);
31    marisa.imparte(ada); ada.impartida(marisa);
32    // Felisa imparte MTP, ADA y POO
33    felisa.imparte(mtp); mtp.impartida(felisa);
34    felisa.imparte(ada); ada.impartida(felisa);
35    felisa.imparte(poo); poo.impartida(felisa);
36    cout << "Personas y sus asignaturas\n"
37         << "--------------------------\n\n";
38    genaro.mostrar();
39    genaro.mostrarAsignaturas();
40    marisa.mostrar();
41    marisa.mostrarAsignaturas();
42    felisa.mostrar();
43    felisa.mostrarAsignaturas();
44    cout << "\nAsignaturas y sus Personas\n"
45         << "--------------------------\n\n";
46    mtp.mostrar();
47    mtp.mostrarPersonas();
48    ada.mostrar();
49    ada.mostrarPersonas();
50    poo.mostrar();
51    poo.mostrarPersonas();
52  }
```

Hay que tener en cuenta que, en muchas ocasiones, la asociación sólo se recorre en un sentido (restricción de navegación), así que sólo es necesario añadir un miembro de datos a una de las clases implicadas en la asociación (la clase origen de la asociación).

Clases de asociación explícitas

Se puede implementar una clase de asociación para una asociación binaria utilizando dos *diccionarios*, uno por cada dirección de la asociación. En el caso de que exista una restricción de navegación basta con un único diccionario.

4.6.2. Implementación de las agregaciones

Para la implementación de las agregaciones se sigue el mismo esquema que con las asociaciones, aunque se puede simplificar de la siguiente forma:

> En el caso de que la existencia de los objetos (componentes) que forman parte de otro (agregado) no tenga sentido independientemente de la existencia de este último, entonces en lugar de utilizar miembros de datos que sean punteros o referencias a los objetos se pueden utilizar los propios objetos.

Ejemplo:

Retómese el supuesto que se propuso en la sección 4.5.3. Supóngase que ya se tiene implementada la clase *Lista* con la siguiente interfaz:

pila/lista.h

```
1  #ifndef LISTA_H_
2  #define LISTA_H_
3
4  #include <deque>
5
6  class Lista {
7  public:
8    bool vacia() const;
9    int primero() const;
10   int ultimo() const;
11   void insertarPrincipio(int e);
12   void insertarFinal(int e);
13   void eliminarPrimero();
14   void eliminarUltimo();
15   void mostrar() const;
16 private:
17   std::deque<int> l;
18 };
19
20 #endif
```

La construcción de la clase *Pila* utilizando agregación puede verse en la cabecera `"pila.h"` a continuación. Nótese cómo se delegan sus operaciones en la clase *Lista*.

pila/pila.h

```
1  #ifndef PILA_H_
2  #define PILA_H_
3
4  #include "lista.h"
5
6  class Pila {
7  public:
8    bool vacia() const;
9    int cima() const;
10   void apilar(int e);
11   void desapilar();
12   void mostrar() const;
13 private:
14   Lista l;
15 };
16
17 // Delegación de operaciones
18
19 inline bool Pila::vacia() const { return l.vacia(); }
20 inline int Pila::cima() const { return l.primero(); }
21 inline void Pila::apilar(int e) { l.insertarPrincipio(e); }
22 inline void Pila::desapilar() { l.eliminarPrimero(); }
23 inline void Pila::mostrar() const { l.mostrar(); }
24
25 #endif
```

4.6.3. Implementación de las generalizaciones

Los lenguajes orientados a objetos proporcionan un mecanismo sencillo para implementar la relación de generalización: la *herencia*.

A diferencia de otros lenguajes, C++ admite la herencia múltiple. En C++ las superclases de una clase se especifican en la declaración de la clase. Las subclases se denominan *clases derivadas* y las superclases, *clases base*.

Los miembros de la clase base son heredados por sus clases derivadas. Se puede acceder a ellos desde cualquier clase derivada, a no ser que se declaren `private`. Los miembros `protected` son accesibles para las clases derivadas pero inaccesibles para las demás.

La accesibilidad a los miembros de una superclase puede restringirse en las clases derivadas durante la herencia. Se puede heredar públicamente, privadamente o protegidamente. Por omisión, se hereda privadamente[13]: si no se especifica otra cosa, los miembros públicos y protegidos de la superclase pasan a ser privados en la clase derivada.

Si una función miembro va a ser reemplazada dinámicamente en alguna clase derivada, entonces hay que declararla como `virtual` en su primera aparición en una clase base.

Las funciones miembro virtuales que se «inicializan a 0» se denominan funciones *virtuales puras* o funciones *abstractas*. Si una clase contiene una función miembro abstracta, se denomina *clase abstracta*, y no se podrán crear directamente objetos de ella. Las funciones miembro abstractas se implementan en las clases derivadas.

Herencia simple

En C++ la sintaxis para derivar una clase de otra ya existente es la siguiente:

```
class clase-derivada: [accesibilidad] clase-base
{
  declaraciones de miembros
};
```

Existen tres tipos de miembros que no se heredan en C++:

- Los constructores.
- Los destructores.
- Los operadores de asignación.

Así, una clase derivada debe proporcionar, si le resulta necesario, sus propias versiones de estos miembros.

Por supuesto, la palabra `class` puede sustituirse por `struct`, cosa que raramente se hace, con el resultado de que los miembros son públicos por omisión, en vez de ser privados de forma predeterminada.

La *accesibilidad*, que es opcional, se especifica mediante una de las tres palabras reservadas `public`, `private` o `protected`. Así se determinan los privilegios de acceso que va a tener la clase derivada sobre los miembros de la clase base (véase la tabla 4.1). Si se omite, ésta se presupone `private` para las clases y `public` para las estructuras.

[13] En `class`. En `struct`, públicamente.

Accesibilidad	un miembro de la clase base...	pasa a ser...
`public`	público	público
	protegido	protegido
	privado	inaccesible
`protected`	público	protegido
	protegido	protegido
	privado	inaccesible
`private`	público	privado
	protegido	privado
	privado	inaccesible

Cuadro 4.1: Cambio de los privilegios de acceso durante la herencia

Mientras que un miembro `private` de una clase únicamente es visible para sus funciones miembro (y sus amigas), uno `protected` lo es también para las inmediatamente derivadas de ella.

Ejemplo:

Supóngase una clase, que se llamará *Base*, con un miembro público, uno privado y otro protegido. Véase qué ocurre cuando se crean clases derivadas de ella de las tres formas posibles.

```
class Base {
public:
  int publico;
protected:
  int privado;
private:
  int protegido;
};

class DerivadaPublica: public Base {
  // ...
};

class DerivadaProtegida: protected Base {
  // ...
};

class DerivadaPrivada: private Base {
  // ...
};
```

En *DerivadaPublica*, *publico* seguirá siendo público y *protegido* seguirá siendo protegido. O sea, al heredar con `public` se conserva la visibilidad de los miembros públicos y protegidos.

En *DerivadaProtegida*, los miembros *publico* y *protegido* heredados pasan a ser protegidos.

Por último, en la clase *DerivadaPrivada*, los miembros *publico* y *protegido* heredados serán ahora miembros privados.

En los tres casos el miembro *privado* de la clase base, que es heredado por las clases derivadas, será inaccesible. Esto es así porque un miembro privado es invisible desde el exterior de la clase a la que pertenece, incluso para sus derivadas. Este miembro privado sólo podrá ser manejado a través de las funciones públicas que la clase base haya dispuesto para ello.

Supóngase que se desea modelar a los estudiantes de un centro universitario como soporte de un sistema de gestión de alumnos que mantenga el registro de éstos, tal y como aparece en la cabecera `"estudiante.h"`:

estudiante/estudiante.h

```
1  #ifndef ESTUDIANTE_H_
2  #define ESTUDIANTE_H_
3
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  class Estudiante {
9  public:
10   Estudiante(string nombre, int dni);
11   void mostrar() const;
12 protected:
13   string nombre;                // nombre completo
14   int dni;                      // DNI
15   // ...
16 };
17
18 #endif
```

Aunque esta clase sería adecuada para estudiantes de primer y segundo ciclo, omite información importante para uno de tercer ciclo (programa de doctorado), como el tutor asignado durante el programa o el código de dicho programa.

A través del mecanismo que proporciona la herencia es posible derivar la clase *Doctorando* a partir de la clase base *Estudiante*, reutilizando su código.

estudiante/doctorando.h

```
#ifndef DOCTORANDO_H_
#define DOCTORANDO_H_

#include "estudiante.h"
#include <string>
using namespace std;

class Doctorando: public Estudiante {
public:
  Doctorando(string nombre, int dni, string tutor, int programa);
  void mostrar() const;
protected:
  string tutor;         // tutor en el programa de doctorado
  int programa;         // código del programa
  // ...
};

#endif
```

estudiante/doctorando.cpp

```
#include "doctorando.h"
#include <iostream>
#include <string>
using namespace std;

Doctorando::
Doctorando(string nombre, int dni, string tutor, int programa):
  Estudiante(nombre, dni), tutor(tutor), programa(programa) {}

void Doctorando::mostrar() const
{
  // Muestra los datos que posee como estudiante
  Estudiante::mostrar();
  // Y los específicos de su condición de doctorando
  cout << "Programa de doctorado: " << programa << "\n"
       << "Tutor en el programa:  " << tutor << endl;
}
```

Obsérvese que el constructor de *Estudiante* se ejecuta como parte de la lista de inicializadores del constructor de *Doctorando*. Esto es usual (¡y aquí sí que es necesaria la sintaxis de inicializadores!), ya que, lógicamente, el subobjeto de la clase base necesita ser construido primero, antes de que el objeto total pueda ser completado.

Como se puede observar, la clase derivada añade nuevos miembros a la base, como ocurre con *tutor* y *programa*, y redefine una función miembro ya existente, *mostrar()*. La función miembro *mostrar()* heredada queda oculta por la nueva. Evidentemente, para que *Doctorando::mostrar()* pueda llamar a la función homónima de la clase base es necesario el operador de resolución de ámbito, o se produciría un bucle infinito.

Véase un programa de prueba:

estudiante/prueba.cpp

```
1  #include "estudiante.h"
2  #include "doctorando.h"
3
4  int main()
5  {
6    Estudiante e("María Pérez Sánchez", 31682034);
7    Doctorando d("José López González", 32456790,
8                 "Dr. Juan Jiménez", 134);
9
10   Estudiante* pe = &e;
11   pe->mostrar();                // e.mostrar()
12   pe = &d;                      // conversión «hacia arriba»
13   pe->mostrar();                // d.estudiante::mostrar()
14   Doctorando* pd = &d;
15   pd->mostrar();                // d.mostrar()
16   pd->Estudiante::mostrar();    // d.estudiante::mostrar()
17 }
```

Al derivar una clase de otra, la relación de generalización establece que *Doctorando* sea un *subtipo* de *Estudiante*. Es decir, un doctorando no es más que un tipo particular de estudiante[14]. De hecho, un objeto de tipo *Doctorando* está formado por un subobjeto de tipo *Estudiante*. Una jerarquía de clases induce así una jerarquía de tipos.

Esto introduce nuevas reglas de tipo. En particular, los objetos de tipos relacionados por la herencia pública son compatibles «hacia arriba»; esto implica que es posible asignar directamente un objeto de tipo *Doctorando* a un *Estudiante*, perdiéndose durante la conversión la información intrínseca al doctorando (sólo se asigna el subobjeto, el resto se elimina).

En consecuencia, un objeto de una clase derivada puede ser tratado en muchos casos como si lo fuera de su clase base (en general, de una clase antecesora), ya que la clase derivada es un subtipo de ella.

[14]Nótese que un estudiante no tiene por qué ser un doctorando.

Por ejemplo, un puntero a la clase base puede apuntar a objetos de la derivada: se produce una conversión del tipo «puntero a la clase derivada» al tipo «puntero a la clase base». Del mismo modo, es posible pasar un objeto de una clase derivada a una función que espera una referencia a un objeto de su clase base. Este tipo de conversiones se denominan «hacia arriba», en alusión al sentido en el que se realizan dentro de la jerarquía de clases.

Véase cómo un *Doctorando* puede convertirse implícitamente en una referencia a *Estudiante*:

```
Doctorando d("José López González", 32456790,
             "Dr. Juan Jiménez", 134);
Estudiante& e = d;
```

La variable *e* es una referencia a *Estudiante*, y la clase base de *d* es *Estudiante*, luego esta conversión es válida, ya que un *Doctorando* es (también) un *Estudiante*.

Sin embargo, un *Estudiante* no es necesariamente un *Doctorando*, por lo que no se puede utilizar un estudiante como si lo fuera:

```
Estudiante e("María Pérez Sánchez", 31682034);
Doctorando& d = e;    // ERROR, conversión incorrecta
```

estudiante/conversiones.cpp

```
1  #include "estudiante.h"
2  #include "doctorando.h"
3
4  int main()
5  {
6    Estudiante e("María Pérez Sánchez", 31682034), *pe;
7    Doctorando d("José López González", 32456790,
8                 "Dr. Juan Jiménez", 134), *pd;
9
10   pe = &d;                                // bien
11   pd = pe;                                // ERROR
12   pd = static_cast<Doctorando*>(pe);      // bien
13   e = d;                                  // bien
14   d = e;                                  // ERROR
15   d = Doctorando(e);                      // ERROR
16   d = static_cast<Doctorando>(e);         // ERROR
17   d = reinterpret_cast<Doctorando>(e); // ERROR
18 }
```

En general, si una clase D hereda públicamente de B, se puede asignar un $D*$ a un $B*$ sin usar conversión explícita de tipos. La conversión inversa, de $B*$ a $D*$, tiene que ser explícita, ya que, en principio, no es segura.

Ejemplo:

Los vectores de bajo nivel que proporciona el lenguaje no hacen comprobaciones sobre los límites. Se va a crear una clase llamada *Vector* que compruebe los límites. Ésta será un pobre sustituto de la clase paramétrica *vector*.

Tras esto se va a reutilizar para crear un vector, igualmente «seguro», cuyos límites inferior y superior sean arbitrarios. Es decir, estos nuevos vectores no empezarán por el índice cero como todos los de C++, sino que se amoldarán al dominio del problema. Por ejemplo, si se quiere medir temperaturas corporales de pacientes lo lógico es utilizar un vector cuyos límites sean, póngase por caso, 34–42 °C.

vector/vector.h

```
1  #ifndef VECTOR_H_
2  #define VECTOR_H_
3
4  class Vector {
5  public:
6    typedef double T;             // tipo base
7    explicit Vector(int n = 10);
8    Vector(const Vector& v);
9    ~Vector();
10   Vector& operator =(const Vector& v);
11   T& operator [](int i);
12   const T& operator [](int i) const;
13   int longitud() const;
14   void mostrar() const;
15 protected:
16   int n;                        // longitud
17   T* v;                         // puntero al primer elemento
18 };
19
20 #endif
```

La clase derivada tendrá además dos miembros privados que guardarán los límites inferior y superior.

vector/supervector.h

```
1  #ifndef SUPERVECTOR_H_
2  #define SUPERVECTOR_H_
3
4  #include "vector.h"
5
6  class SuperVector: public Vector {
```

```
7  public:
8    explicit SuperVector(int inferior = 0, int superior = 9);
9    T& operator [](int i);
10   const T& operator [](int i) const;
11   int limiteInferior() const;
12   int limiteSuperior() const;
13 private:
14   int inferior,                 // límite inferior
15       superior;                 // límite superior
16 };
17
18 #endif
```

vector/supervector.cpp

```
1  #include "supervector.h"
2  #include <cassert>
3
4  SuperVector::SuperVector(int inferior, int superior):
5    Vector(superior - inferior + 1),
6    inferior(inferior), superior(superior)
7  {}
8
9  // Operadores de índice
10
11 SuperVector::T& SuperVector::operator [](int i)
12 {
13   assert(i >= inferior && i <= superior);
14   return Vector::operator [](i - inferior);
15 }
16
17 const SuperVector::T& SuperVector::operator [](int i) const
18 {
19   assert(i >= inferior && i <= superior);
20   return Vector::operator [](i - inferior);
21 }
22
23 // Límites
24
25 int SuperVector::limiteInferior() const
26 {
27   return inferior;
28 }
29
30 int SuperVector::limiteSuperior() const
31 {
32   return superior;
33 }
```

Nótese que, pese a que no se heredan, no es necesario definir constructor de copia, ni destructor, ni operador de asignación. Todas estas operaciones se realizan miembro a miembro *incluyendo al subobjeto heredado*, por lo que se ejecutarán sus contrapartidas de la clase base cuando sea necesario.

En la sobrecarga del operador índice (corchetes) se comprueban los rangos y, si están bien, se reutiliza el código de la clase base llamando a su operador.

Esto funciona, pero es ineficiente porque se supone que el operador índice de la clase *Vector* también comprueba los límites (recuérdese que dicha clase tenía la propiedad de que se comprobaban los límites de los vectores). Así que se comprueban los límites dos veces. ¿Cómo se puede solucionar? Cambiando el cuerpo del operador a:

```
assert(i >= inferior && i <= superior);
return v[i - inferior];
```

Nótese que para esto es necesario que v sea `protected`. Para esto se inventó esta nueva palabra reservada: un miembro protegido sólo puede ser accedido desde fuera de su clase por una derivada. El que un miembro sea privado o protegido dependerá de que sea potencialmente reutilizable.

Herencia múltiple

Los ejemplos vistos hasta ahora han requerido herencia simple; o sea, que una clase se derive de una sola clase base. Esto puede llevar a una cadena de derivaciones donde la clase B se deriva de A, C de B, ..., y la clase N deriva de M. Al final N acaba estando basada en A, B, ..., M. Esta cadena no debe ser circular, pues una clase no puede tenerse a sí misma como ancestro.

La herencia múltiple permite que una clase derive de más de una clase base. Así se intentan combinar distintas características de clases diferentes para formar una clase nueva diferenciada.

La sintaxis se extiende de manera natural para permitir especificar una lista de clases bases y sus restricciones de accesibilidad.

Ejemplo:

```
class Empleado {
  // ...
};

class Administrador: public Empleado {
```

```
protected:
  Nivel nivel;
  // ...
};

class Director: public Administrador {
  // ...
};
```

donde *nivel* es un objeto de otra clase *Nivel*.

Un conjunto así de clases relacionadas se denomina *jerarquía de clases*. La jerarquía es con mayor frecuencia un árbol, pero no necesariamente:

```
class Temporal {
  // ...
};

class Secretaria: public Empleado {
  // ...
};

class SecretariaTemporal: public Temporal, public Secretaria {
  // ...
};

class Asesor: public Temporal, public Administrador {
  // ...
};
```

Una relación de parentesco como ésta se describe por un grafo dirigido acíclico. En esta estructura los vértices son clases y las aristas van desde las clases bases a las derivadas. No puede ser circular.

Orden de inicialización

Si se define un objeto de la clase *Asesor*, los constructores se ejecutan en el siguiente orden:

1. Los de la clase base en el orden en que han sido declarados en la lista de derivación:
   a) *Temporal()*.
   b) *Empleado()*, por ser clase base de *Administrador*.

   c) *Administrador()*, que llamará a *Nivel()*.

2. Cada miembro de la clase en el orden en que ha sido declarado dentro de la clase (independientemente del orden de la lista de inicialización del constructor)[15].

3. El constructor de la clase derivada, *Asesor()*.

Los destructores se ejecutan en orden inverso.

Ambigüedades inherentes

En la herencia múltiple es normal que surjan ambigüedades; por ejemplo, si se tiene dos clases base con algunos miembros iguales y una derivada de ellas.

Ejemplo:

ambiguedades/conflicto.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class B1 {
5  public:
6    void f() { cout << "B1::f()" << endl; }
7    int b;
8    // ...
9  };
10
11 class B2 {
12 public:
13   void f() { cout << "B2::f()" << endl; }
14   int b;
15   // ...
16 };
17
18 class D: public B1, public B2 {
19   // ...
20 };
21
22 int main()
23 {
24   D d;
25   d.f();          // ERROR, ¿qué f(), el de B1 o el de B2?
```

[15]Por claridad, conviene poner la lista en el mismo orden.

```
26    d.b = 0;         // ERROR, ¿qué «b», el de B1 o el de B2?
27    d.B1::f();       // bien
28    d.B2::f();       // bien
29    d.B1::b = 0;     // bien
30    d.B2::b = 0;     // bien
31  }
```

Como se ve, el problema está en que las clases base utilizan miembros con identificadores iguales. El error se produciría incluso si uno fuera público y otro privado o protegido. La solución está en utilizar el operador de resolución de ámbito, o crear un miembro con ese nombre en la clase derivada.

Observe que el miembro ambiguo *b* forma parte de la clase derivada por partida doble; es decir, el tamaño de un objeto de la clase derivada sería la suma de los tamaños de las dos clases base más el aportado por la propia clase derivada.

También hay que tener en cuenta que la resolución de la sobrecarga no se aplica entre ámbitos diferentes. En particular, las ambigüedades entre funciones miembro de clases base diferentes no se resuelven de acuerdo con los tipos de los parámetros.

Ejemplo:

ambiguedades/sobrecarga.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   class B1 {
5   public:
6     void f(int i) { cout << "B1::f(int)" << endl; }
7     // ...
8   };
9
10  class B2 {
11  public:
12    void f(double d) { cout << "B2::f(double)" << endl; }
13    // ...
14  };
15
16  class D: public B1, public B2 {
17    // ...
18  };
19
20  int main()
```

```
21  {
22    D d;
23    d.f(0);          // ERROR, ¿qué f(), el de B1 o el de B2?
24    d.f(0.0);        // ERROR, ¿qué f(), el de B1 o el de B2?
25    d.B1::f(0);      // bien, B1::f()
26    d.B2::f(0.0);    // bien, B2::f()
27  }
```

Cuando se combinan clases que esencialmente no guardan relación entre sí, como *B1* y *B2*, se produce un conflicto de nombres como se puede ver en el ejemplo anterior.

En el caso de que se quiera que la selección esté basada en los tipos de los parámetros se utilizarán declaraciones `using` para llevar las funciones a un ámbito común.

Ejemplo:

ambiguedades/using.cpp

```
1   #include <iostream>
2   using namespace std;
3
4   class B1 {
5   public:
6     void f(char i) { cout << "B1::f(char)" << endl; }
7     // ...
8   };
9
10  class B2 {
11  public:
12    void f(int d) { cout << "B2::f(int)" << endl; }
13    // ...
14  };
15
16  class D: public B1, public B2 {
17  public:
18    using B1::f;
19    using B2::f;
20    void f(double c) { cout << "D::f(double)" << endl; }
21    // ...
22  };
23
24  int main()
25  {
26    D d;
```

```
27    d.f('A');     // B1::f(char)
28    d.f(0);       // B2::f(int)
29    d.f(0.0);     // D::f(double)
30  }
```

Estas declaraciones `using` han sido incorporadas recientemente al lenguaje, por lo que la mayoría de los compiladores actuales no las permiten.

Otro problema que puede aparecer con la herencia múltiple es la posibilidad de heredar un mismo miembro por caminos diferentes. Esto se conoce como *herencia duplicada*. Para verlo con un ejemplo, se va a utilizar el caso más simple, donde cada clase tiene un miembro de datos público (un entero):

```
struct A { int a; };
struct B: A { int b; };
struct C: A { int c; };
struct D: B, C { int d; };
```

La clase D va a tener un miembro d, evidentemente, pero por derivar de B tendrá un miembro b y, de la misma manera, otro c; ahora bien, como tanto B como C derivan de A, D tendrá *dos* miembros a. Esto produce, de momento, una ambigüedad:

```
D d;
d.a = 0;       // ERROR, ambigüedad
d.B::a = 0;    // bien, resolución de la ambigüedad
d.C::a = 0;    // bien, resolución de la ambigüedad
```

Pero lo más importante es que normalmente sólo se quiere una copia de a y no dos. Para solventar esto, en C++ se utiliza la *herencia virtual*, que consiste en nuestro ejemplo en heredar virtualmente la clase A dentro de B y C. Así, se pondría:

```
struct B: virtual A { int b; };
struct C: virtual A { int c; };
```

4.7. Polimorfismo en tiempo de ejecución

La herencia y el polimorfismo constituyen la parte fundamental de la programación orientada a objetos. La herencia permite que unas clases se puedan

construir a partir de otras, y el polimorfismo, tratar objetos de clases relacionadas de una forma genérica.

C++ admite métodos *virtuales* a través de la palabra reservada `virtual`. Estos métodos se declaran en la clase base y se redefinen en la derivada. Una jerarquía de clases definida mediante herencia crea un conjunto relacionado de tipos cuyos objetos pueden ser apuntados por un puntero a la clase base. Accediendo al método *virtual* a través de este puntero, C++ selecciona la definición del método apropiado en tiempo de ejecución.

Funciones miembro virtuales

Cuando se llama a un método sobrecargado, un algoritmo se encarga de seleccionar el método adecuado según el número y tipo de los parámetros; esto incluye que el parámetro implícito (el objeto al que se aplica el método) concuerde con el tipo de la clase correspondiente. Todo esto es conocido en tiempo de compilación (*static binding*, enlace estático) y por tanto es el compilador el que directamente selecciona el método apropiado. Considere el ejemplo de los estudiantes y doctorandos de la sección 4.6.3. Observe cómo el método *mostrar()* llamado se selecciona en tiempo de compilación según el tipo del objeto.

Sin embargo quizá vaya siendo evidente que estaría bien que el método apropiado fuera seleccionado en tiempo de ejecución (*dynamic binding*, enlace dinámico) entre los de la clase base o derivada. Para esto existe la palabra reservada `virtual`, que es un especificador de función que se aplica sólo a declaraciones de métodos. Esto es una forma de *polimorfismo puro*. Como quizá ya sepa, el polimorfismo, que es uno de los pilares sobre los que se asienta la POO, consiste en que una misma «llamada a método» (*mensaje*) sirve para diferentes clases y se comporta de diferente forma según el objeto sobre el que actúa; o sea, un mismo mensaje puede tener varias (*poli*) formas (*morfismo*) en función del objeto.

Un método *virtual* ordinario debe ser definido con código ejecutable como otro cualquiera. Cuando se llama, su semántica es también como la de cualquiera. En una clase derivada puede ser definido de otra forma, aunque su *signatura* y tipo de retorno deben coincidir. Como se ha dicho, la selección del método a ser llamado es dinámica, en tiempo de ejecución.

El caso típico ocurre cuando una clase base tiene un método virtual y las clases derivadas tienen su propia versión de ese método. Como ya se sabe, un puntero a la clase base puede apuntar a un objeto tanto de la clase base como de una derivada. Si se llama al método a través de tal puntero, la selección dependerá no del tipo del puntero como en el ejemplo anterior de los estudiantes, sino del objeto al que apunte. En caso de ausencia de un

miembro de tipo derivado se llamaría al método virtual de la clase base. Véase un pequeño ejemplo ilustrativo.

virtual/polipur.cpp

```
1  // Ejemplo de selección de función virtual
2
3  #include <iostream>
4  using namespace std;
5
6  class B {
7  public:
8    virtual void mostrar() { cout << i << " dentro de B\n"; }
9    int i;
10 };
11
12 class D: public B {
13 public:
14   void mostrar() { cout << i << " dentro de D\n"; }
15 };
16
17 int main()
18 {
19   B b, *pb = &b;
20   D d;
21
22   d.i = 1 + (b.i = 1);  // b.i = 1, d.i = 1 + 1;
23   pb->mostrar();        // B::mostrar()
24   pb = &d;
25   pb->mostrar();        // D::mostrar()
26 }
```

La salida de este programa deberá ser:

```
1 dentro de B
2 dentro de D
```

Usando la terminología de la POO, se ha enviado el mensaje *mostrar()* al objeto y éste ha seleccionado su propia versión del método correspondiente. Así, el tipo base del puntero no ha sido lo determinante en la selección del método.

Observe que esta selección no es la misma que la de un método sobrecargado normal. Éste sería seleccionado según su signatura en tiempo de compilación, y el tipo de retorno podría ser distinto. En cambio un método virtual se selecciona en tiempo de ejecución según el tipo del objeto que le es pasado como parámetro implícito; esto es, como el puntero `this`. Además, una vez

declarado `virtual`, esta propiedad se propaga a todas las redefiniciones de las clases derivadas, que no tienen que usar ya este modificador.

La mezcla de métodos virtuales y sobrecargados da problemas y produce confusión.

Ejemplo:

virtual/virtual.cpp

```
1  // Ejemplo de selección de función virtual y sobrecargada
2
3  #include <iostream>
4  using namespace std;
5
6  class B {
7  public:
8    virtual void mostrar(int i)
9      { cout << i << " dentro de B::mostrar(int)\n"; }
10   virtual void mostrar(double d)
11     { cout << d << " dentro de B::mostrar(double)\n"; }
12   virtual void mostrar(char c)
13     { cout << c << " dentro de B::mostrar(char)\n"; }
14 };
15
16 class D: public B {
17 public:
18   virtual void mostrar(int i)
19     { cout << i << " dentro de D::mostrar(int)\n"; }
20 };
21
22 int main()
23 {
24   D d;
25   B b, *pb = &d;
26
27   b.mostrar(9);       // B::mostrar(int)
28   b.mostrar(9.5);     // B::mostrar(double)
29   b.mostrar('a');     // B::mostrar(char)
30   d.mostrar(9);       // D::mostrar(int)
31   d.mostrar(9.5);     // D::mostrar(int)
32   d.mostrar('a');     // D::mostrar(int)
33   pb->mostrar(9);     // D::mostrar(int)
34   pb->mostrar(9.5);   // B::mostrar(double)
35   pb->mostrar('a');   // B::mostrar(char)
36 }
```

El método de la clase base `B::mostrar(int)` se sustituye en la clase derivada.

`B::mostrar(double)`, en cambio, se *oculta*. En realidad, en la instrucción `d.mostrar(9.5)` el valor `double` 9.5 se convierte en un entero y se trunca a 9. Para llamar al método oculto se debería haber puesto `d.B::mostrar(9.5)`.

Como se sabe, la declaración de cualquier identificador en un cierto ámbito oculta todas sus declaraciones anteriores hechas en ámbitos exteriores. Lo que realmente ocurre es que una clase base *es* un ámbito superior para una derivada; esta regla es independiente de si los métodos se declaran `virtual` o no.

Algunos puntos más a tener en cuenta:

- Para conseguir un comportamiento polimórfico, las funciones miembros llamadas deben ser `virtual` y los objetos deben ser manipulados mediante punteros o referencias.

  Cuando se manipula un objeto directamente (en lugar de hacerlo mediante punteros o referencias) su tipo exacto es conocido en tiempo de compilación, por lo que no es necesario polimorfismo de tiempo de ejecución.

- Llamar a una función utilizando el operador de resolución de ámbito asegura que no se emplea el mecanismo virtual.

- Sólo métodos no estáticos pueden ser virtuales.

- La característica `virtual` se hereda, por lo que no hace falta ponerla en las clases derivadas.

- Los constructores no pueden ser virtuales.

- Los destructores sí pueden ser virtuales.

- Como regla general, una clase que tenga métodos virtuales debe tener un destructor virtual, aunque esté vacío. Así se elegirá en tiempo de ejecución.

Los métodos virtuales permiten decisiones en tiempo de ejecución.

Ejemplo:

Supóngase el siguiente ejemplo. En una aplicación de CAD se necesita hallar el área de ciertas figuras en un diseño. Las figuras derivan de la clase base `Figura`:

area/figura.h

```
#ifndef FIGURA_H_
#define FIGURA_H_

class Figura {
public:
  virtual ~Figura() {};
  virtual double area() const { return 0.0; } // por omisión
  // ...
};

class Rectangulo: public Figura {
public:
  Rectangulo(double lado_1, double lado_2);
  // ...
  double area() const;
protected:
  double lado_1, lado_2;
};

class Circulo: public Figura {
public:
  Circulo(double radio);
  // ...
  double area() const;
private:
  double radio;
};

// ...

#endif
```

Como se ve, es muy fácil añadir nuevas figuras. El cálculo del área es responsabilidad de la clase derivada. Un fragmento de código cliente que calculara el área total polimórficamente sería algo así:

```
vector<Figura*> figura;
// ...
double area = 0.0;
for (size_t i = 0; i < figuras.size(); ++i)
  area += figura[i]->area();
```

Se define *figura* como un vector de punteros a `Figura`. En los puntos suspensivos se ha asignado cada elemento —esto es, cada puntero a `Figura`— a una figura distinta; por ejemplo, `figura[2]` podría apuntar a un objeto

círculo y `figura[4]` a un rectángulo. Así, `figura[2]->area()` calcularía el área de un círculo llamando a *Circulo::area()*, etc.

Una ventaja de esto es que este código cliente no tiene que cambiar si se añaden nuevas figuras.

Clases abstractas

En una jerarquía de tipos es normal que la clase principal, o sea la raíz, la base de las demás, contenga cierto número de métodos virtuales, para proporcionar lo que se ha visto antes. Suelen tener un cuerpo vacío en dicha clase raíz, puesto que los significados específicos se dan en cada una de las clases derivadas. Es algo parecido a lo que se ha visto antes con las figuras, aunque ahí al método virtual de la clase base se le hizo devolver un área cero.

Para casos de éstos, en C++ existe el llamado *método virtual puro*. En éste, el cuerpo ni siquiera se define; se declara dentro de la clase de la siguiente extraña manera:

```
virtual prototipo = 0;
```

Se usa para diferir la definición del método; en POO se dice que este método es un *método diferido*; como si se dijera, «ya se definirá luego en otro sitio».

Una clase que tiene al menos un método virtual puro es una *clase abstracta*. Es útil a veces que la clase raíz de una jerarquía sea abstracta; esto implica que tiene las propiedades comunes de sus clases derivadas, pero que no puede utilizarse ella misma para declarar objetos de su tipo, sino más bien punteros a él, que luego podrán apuntar a objetos de las clases derivadas.

Ejemplo:

Se va a rehacer y completar el ejemplo de las figuras geométricas de la sección anterior haciendo abstracta la clase raíz.

figura/figura.h

```
1  #ifndef FIGURA_H_
2  #define FIGURA_H_
3
4  class Figura {
5  public:
```

```
6     virtual ~Figura();
7     virtual double area() const = 0;
8     // ...
9     virtual void mostrar() const = 0;
10  };
11
12  // Destructor virtual vacío
13
14  inline Figura::~Figura() {}
15
16  #endif
```

figura/circulo.h

```
1   #ifndef CIRCULO_H_
2   #define CIRCULO_H_
3
4   #include <iostream>
5   #include <cmath>
6
7   class Circulo: public Figura {
8   public:
9     Circulo(double radio);
10    // ...
11    double area() const;
12    void mostrar() const;
13  private:
14    double radio;
15  };
16
17  inline Circulo::Circulo(double radio):
18    radio(radio) {}
19
20  inline double Circulo::area() const
21  {
22    static const double pi = 4.0 * std::atan(1.0);
23    return pi * radio * radio;
24  }
25
26  inline void Circulo::mostrar() const
27  {
28    std::cout << "Círculo(" << radio << ")";
29  }
30
31  #endif
```

figura/rectangulo.h

```
#ifndef RECTANGULO_H_
#define RECTANGULO_H_

#include <iostream>

class Rectangulo: public Figura {
public:
  Rectangulo(double lado_1, double lado_2);
  double area() const;
  // ...
  void mostrar() const;
protected:
  double lado_1, lado_2;
};

inline Rectangulo::Rectangulo(double lado_1, double lado_2):
  lado_1(lado_1), lado_2(lado_2) {}

inline double Rectangulo::area() const
{
  return lado_1 * lado_2;
}

inline void Rectangulo::mostrar() const
{
  std::cout << "Rectángulo(" << lado_1 << ", " << lado_2 << ")";
}

#endif
```

figura/cuadrado.h

```
#ifndef CUADRADO_H_
#define CUADRADO_H_

#include <iostream>

class Cuadrado: public Rectangulo {
public:
  Cuadrado(double lado);
  // ...
  void mostrar() const;
};

inline Cuadrado::Cuadrado(double lado):
  Rectangulo(lado, lado) {}

```

```
16 inline void Cuadrado::mostrar() const
17 {
18   std::cout << "Cuadrado(" << lado_1 << ")";
19 }
20
21 #endif
```

figura/prueba-1.cpp

```
1  #include <cstdlib>
2  #include <ctime>
3  #include "figura.h"
4  #include "rectangulo.h"
5  #include "cuadrado.h"
6  #include "circulo.h"
7  using namespace std;
8
9  Figura* figura_aleatoria();
10
11 int main()
12 {
13   srand(time(0));
14
15   for (int i = 0; i < 10; ++i) {
16     Figura* f = figura_aleatoria();
17
18     cout << "Figura = ";
19     f->mostrar();
20     cout << "      \t" << "Área = " << f->area() << endl;
21     delete f;
22   }
23 }
24
25 // Genera una figura aleatoria (rectángulo, cuadrado o círculo)
26
27 Figura* figura_aleatoria()
28 {
29   double x = rand() % 9 + 1, y = rand() % 9 + 1;
30   switch (rand() % 3) {
31   case 0:
32     return new Rectangulo(x, y);
33   case 1:
34     return new Cuadrado(x);
35   default:
36     return new Circulo(x);
37   }
38 }
```

figura/prueba-2.cpp

```
#include <cstdlib>
#include <ctime>
#include "figura.h"
#include "rectangulo.h"
#include "cuadrado.h"
#include "circulo.h"
using namespace std;

Figura& figura_aleatoria();

int main()
{
  srand(time(0));

  // «f» es un objeto polimórfico

  for (int i = 0; i < 10; ++i) {
    Figura& f = figura_aleatoria();

    cout << "Figura = ";
    f.mostrar();
    cout << "      \t" << "Área = " << f.area() << endl;
    delete &f;
  }
}

// Genera una figura aleatoria (rectángulo, cuadrado o círculo)

Figura& figura_aleatoria()
{
  double x = rand() % 9 + 1, y = rand() % 9 + 1;
  switch (rand() % 3) {
  case 0:
    return *new Rectangulo(x, y);
  case 1:
    return *new Cuadrado(x);
  default:
    return *new Circulo(x);
  }
}
```

4.7.1. Identificación de tipos en tiempo de ejecución

C++ posee un mecanismo denominado RTTI (*run-time type identification*) que permite determinar con seguridad, a partir de un puntero o referencia de tipo polimórfico, el tipo real del objeto al que se hace referencia.

Parte de este sistema se implementa a través del operador de modelado `dynamic_cast`, que se describirá brevemente a continuación. Este operador es paramétrico y adopta la siguiente forma:

```
dynamic_cast<tipo>(valor)
```

donde *tipo* debe ser un tipo puntero o referencia y *valor* un valor apropiado para su conversión a dicho tipo.

Si la conversión es válida, se devuelve el valor del tipo requerido. En caso contrario, se devuelve un 0 (el puntero nulo) si se solicitó la conversión a un tipo puntero, o se lanza una excepción estándar de tipo *bad_cast* si el tipo de destino era una referencia.

Ejemplo:

Supóngase los siguientes tipos polimórficos:

```
class B {
public:
  virtual ~B() {}
  // ...
};

class D: public B {
  //...
};
```

La siguiente función procesa a través de un puntero un objeto perteneciente a la jerarquía de clases de raíz B. Sin embargo, trata especialmente los que son en realidad de tipo D.

```
void procesar(B* pb)
{
  if (D* pd = dynamic_cast<D*>(pb)) {
    // El objeto apuntado por «pb» es de tipo «D»
    // Así «pb» se convierte sin problemas en «pd»
  }
  else {
```

```
    // El objeto apuntado por «pb» no es de tipo «D»
    // La conversión ha fallado
  }
}
```

Nótese la diferencia entre `dynamic_cast` y un modelado ordinario: en general no es seguro realizar un modelado ordinario ya que se está modelando «hacia abajo» (¿y si apuntara realmente a un objeto que no es de tipo *D* ni de ninguno de sus tipos derivados?).

Operador `typeid`

La otra forma de emplear la RTTI consiste en un operador como `sizeof`, en el sentido de que parece una función, pero está implementado directamente en el compilador. El operador se llama `typeid` y su operando, que va entre paréntesis, es un objeto, una referencia, un puntero o un nombre de tipo; y devuelve una referencia a un objeto global no modificable de tipo *type_info*. Este tipo es una clase definida en la cabecera `<typeinfo>`. Los objetos de esta clase se pueden comparar mediante los operadores `==` y `!=`, se puede obtener el nombre (una representación como cadena de caracteres dependiente de la implementación) mediante el método *name()* y se pueden clasificar mediante el método *before()*.

Ejemplo:

En este ejemplo, *Forma* es una clase base de *Circulo*, *Cuadrado*, etc. No hay que incluir `<typeinfo>` puesto que no se emplea ningún método ni operador de la clase *type_info*.

```
void f(Forma& r, Forma* p)
{
  typeid(r);  // tipo del objeto referido por 'r'
  typeid(*p); // tipo del objeto al que apunte 'p'
  typeid(p);  // tipo Forma*; válido pero evidente
}
```

Si el valor de un operando puntero o referencia es 0, `typeid` lanza una excepción de tipo *bad_typeid*.

Por consistencia, `typeid` funciona con los tipos incorporados en el lenguaje, pero sólo tiene verdadero sentido con tipo polimórficos. Asimismo, ni `typeid`

ni `dynamic_cast` funcionarán con punteros genéricos, pues el compilador no tiene información de su tipo base real.

EJEMPLO:

Un uso simple de `typeid` es el de mostrar a los estudiantes cómo funciona el lenguaje. En el siguiente ejemplo se muestra el orden de llamadas a constructores y destructores sin emplear macros del preprocesador[16].

orden.cpp

```
1  /* Orden de llamadas a constructores
2   * Ejemplo de uso de typeid
3   */
4  #include <iostream>
5  #include <typeinfo>
6  using namespace std;
7
8  template<int id> class Anuncio {
9  public:
10   Anuncio() { cout << "__" << typeid(*this).name() << endl; }
11   ~Anuncio() { cout << "~__" << typeid(*this).name() << endl; }
12 };
13
14 class X: public Anuncio<0> {
15   Anuncio<1> m1;
16   Anuncio<2> m2;
17 public:
18   X() { cout << "Constructor de X::X()" << endl; }
19   ~X() { cout << "Destructor de X::~X()" << endl; }
20 };
21
22 int main()
23 {
24   X x;
25 }
```

Hay que incluir la cabecera `<typeinfo>` puesto que se llama al método *name()*. Se emplea una plantilla con un parámetro *int* para diferenciar una clase de otra. Dentro del constructor y del destructor se emplea la información de RTTI para producir el nombre de la clase. La clase *X* emplea tanto herencia como composición para mostrar un interesante orden de llamadas a constructor y destructor.

[16]En LINUX, ejecútese el programa pasándole el filtro `c++filt`; por ejemplo: `orden | c++filt`, para obtener un resultado legible, pues el compilador cambia el nombre de las funciones de las plantillas.

4.8. Plantillas

Describiremos a continuación el empleo de *plantillas* (en inglés, *templates*). Este nuevo concepto nos conduce a una variante del polimorfismo: el *polimorfismo paramétrico* o *polimorfismo en tiempo de compilación*. Principalmente nos centraremos en el estudio de las clases paramétricas, ya que las funciones y operadores miembro se parametrizan análogamente a los que no lo son.

La posibilidad de emplear un tipo de datos como parámetro refuerza el hecho de que C++ sea un lenguaje apropiado para la implementación de especificaciones formales a partir de un TAD. Un TAD paramétrico se implementa en C++ directamente a través de una clase paramétrica.

El concepto es similar al que proporcionan las *variables de tipo* que aparecen en otros lenguajes y permite llevar a término el paradigma de la programación genérica.

Con el empleo de plantillas se consigue un código más reutilizable, fácil de mantener, e independiente de los detalles internos.

4.8.1. Clases paramétricas

Suele ocurrir en lenguajes que no poseen parametrizaciones de tipo (como es el caso de C) que un programador reimplementa una y otra vez un mismo tipo abstracto de datos con ligeras variantes.

Esto es más acusado aún en cuanto a tipos contenedores se refiere, pues es a menudo frecuente la necesidad de disponer de listas, pilas, colas, árboles y demás estructuras de datos para diversos tipos base. El programador acaba pues, programando una gran cantidad de código que se diferencia únicamente en pequeños matices, muchas veces únicamente en los tipos subyacentes.

Esta tarea es tediosa y propensa a errores, el programador tiende a «cortar y pegar» código de proyectos anteriores y no es difícil que olvide alguno de los pequeños cambios que debe realizar. Rara vez cae en la necesidad de invertir algo de esfuerzo en disponer de una biblioteca reutilizable capaz de manejar tales estructuras con una mínima dependencia del tipo base.

> Una forma de hacer esto en C consiste en emplear punteros genéricos (*void **) para que las funciones puedan recibir punteros a elementos de cualquier tipo (es una forma de hacer que se pase por alto la comprobación de tipos). Todo habría de ser manejado por el propio programador que tendría que encargarse de que cada puntero genérico fuera acompañado por información explícita sobre el tipo correcto correspondiente al objeto apuntado.

Las clases paramétricas de C++ permiten definir clases contenedoras reutilizables una vez acordados los requisitos que deben cumplir los objetos contenidos.

Por ejemplo, es posible crear una clase *Pila*[17] independiente prácticamente del tipo base. Es en la definición de los objetos (variables de la clase *Pila*) donde se especifica el tipo concreto:

```
Pila<char> p1;                      // pila de caracteres
Pila<string> p2(200);               // pila de cadenas
Pila<complex<double> > p3(300);     // pila de complejos
```

En las definiciones *especializamos* la clase genérica *Pila* para conseguir las clases *Pila*<*char*>, *Pila*<*string*> y *Pila*<*complex*<*double*> >[18]. Decimos que estos tres tipos son *especializaciones* del tipo paramétrico *Pila*.

En C++ todo es mucho más sencillo, los detalles internos son manejados por el propio compilador, dejando al programador abstraerse de todos estos problemas y centrarse en la creación del nuevo tipo de datos.

Definición

Para definir una clase paramétrica C que dependa de un tipo T se antepone el prefijo `template <typename T>` a lo que sería su definición habitual. La palabra reservada `typename` puede ser sustituida por `class`, sin que esto indique que el tipo parámetro deba ser una clase, de hecho, puede ser cualquier tipo.

> La palabra `class` se emplea de manera habitual en este contexto, en sustitución de `typename`. Esto se debe principalmente, aparte de por ser más corta, a que históricamente `typename` se introdujo después que las plantillas.

Una vez hecho esto, el identificador T puede emplearse dentro de la clase como si fuera un tipo cualquiera. El nombre de la clase paramétrica es C<T>, aunque dentro del ámbito de la clase puede abreviarse a C.

Si una función u operador paramétrico se define fuera de una clase, la sintaxis es similar.

[17] De hecho, la biblioteca de C++ ya incorpora una.

[18] Nótese el espacio: *Pila*<*complex*<*double*>> produciría un error sintáctico, ya que >> forma un lexema, el del operador de desplazamiento de bits a la derecha.

Ejemplo:

Veamos cómo es posible implementar matrices paramétricas con comprobación de rango reutilizando mediante agregación una clase *Vector* que crearemos al efecto basándonos en los vectores estándares.

La clase *Vector* se define a partir de *vector* como una clase paramétrica dependiente de un cierto tipo base T y posee:

1. Un constructor que permite crear un vector de la dimensión indicada inicializando cada elemento con un mismo valor del tipo apropiado. Sus parámetros son omitibles apareciendo dos constructores adicionales:
   a) Un constructor explícito que crea un vector de la dimensión indicada inicializando cada elemento con el constructor por omisión del tipo base.
   b) Un constructor por omisión que crea un vector de un solo elemento inicializado con el constructor por omisión del tipo base.
2. Operadores de índice: uno, accesor, para vectores normales y otro, observador, para vectores constantes.
3. Operadores miembro para realizar la auto-suma, la auto-resta y el auto-producto externo (multiplicación de los elementos del vector por un valor del tipo parámetro).
4. Operadores externos para realizar la suma, la resta y el producto externo (dos versiones: por la izquierda y por la derecha).
5. Una función miembro observadora que devuelve la dimensión de un vector.
6. Una función miembro observadora que muestra un vector.

vector.h

```
1  #ifndef VECTOR_H_
2  #define VECTOR_H_
3
4  #include <cassert>
5  #include <vector>
6  using std::vector;
7
8  template <typename T> class Vector {
9  public:
10   explicit Vector(size_t n = 1, T x = T());
11   T& operator [](size_t i);
```

```
12    const T& operator [](size_t i) const;
13    Vector& operator +=(const Vector& a);
14    Vector& operator -=(const Vector& a);
15    Vector& operator *=(const T& k);
16    size_t dimension() const;
17    void mostrar() const;
18  protected:
19    vector<T> v;                  // elementos
20  };
21
22  // Constructores
23
24  template <typename T> inline
25  Vector<T>::Vector(size_t n, T x): v(n, x) {}
26
27  // Dimensión
28
29  template <typename T> inline
30  size_t Vector<T>::dimension() const { return v.size(); }
31
32  // Operadores de índice
33
34  template <typename T> inline
35  T& Vector<T>::operator [](size_t i)
36  {
37    assert(i < dimension());
38    return v[i];
39  }
40
41  template <typename T> inline
42  const T& Vector<T>::operator [](size_t i) const
43  {
44    assert(i < dimension());
45    return v[i];
46  }
47
48  // Operador de suma
49
50  template <typename T> inline
51  Vector<T> operator +(const Vector<T>& a, const Vector<T>& b)
52  {
53    assert(a.dimension() == b.dimension());
54    return Vector<T>(a) += b;
55  }
56
57  // Operador de resta
58
59  template <typename T> inline
60  Vector<T> operator -(const Vector<T>& a, const Vector<T>& b)
```

```
61  {
62    assert(a.dimension() == b.dimension());
63    return Vector<T>(a) -= b;
64  }
65
66  // Operadores de producto externo
67
68  template <typename T> inline
69  Vector<T> operator *(const Vector<T>& a, const T& k)
70  {
71    return Vector<T>(a) *= k;
72  }
73
74  template <typename T> inline
75  Vector<T> operator *(const T& k, const Vector<T>& a)
76  {
77    return a * k;
78  }
79
80  // Definiciones de plantillas no «inline»
81
82  #include "vector.cpp"
83
84  #endif
```

Nótese la inclusión del fichero que contiene ¡la definición de las plantillas! Hasta la fecha pocos compiladores de C++ implementan un buen mecanismo de especialización automática de plantillas. Este método, tan poco elegante, siempre funciona: se trata de incluir el código que define la clase paramétrica en la misma unidad de traducción en la que se usa, justo antes de su utilización.

En §4.8.4 se explicarán otras posibilidades.

vector.cpp

```
1   #include "vector.h"
2   #include <iostream>
3   using namespace std;
4
5   // Operador de auto-suma
6
7   template <typename T>
8   Vector<T>& Vector<T>::operator +=(const Vector<T>& a)
9   {
10    assert(dimension() == a.dimension());
11    for (size_t i = 0; i < dimension(); ++i)
12      (*this)[i] += a[i];
```

```
13    return *this;
14  }
15
16  // Operador de auto-resta
17
18  template <typename T>
19  Vector<T>& Vector<T>::operator -=(const Vector<T>& a)
20  {
21    assert(dimension() == a.dimension());
22    for (size_t i = 0; i < dimension(); ++i)
23      (*this)[i] -= a[i];
24    return *this;
25  }
26
27  // Operador de auto-producto externo
28
29  template <typename T>
30  Vector<T>& Vector<T>::operator *=(const T& k)
31  {
32    for (size_t i = 0; i < dimension(); ++i)
33      (*this)[i] *= k;
34    return *this;
35  }
36
37  // Mostrar
38
39  template <typename T>
40  void Vector<T>::mostrar() const
41  {
42    for (size_t i = 0; i < dimension(); ++i)
43      cout << (*this)[i] << ' ';
44    cout << endl;
45  }
```

Cuando se define una clase paramétrica como *Vector* se están realizando implícitamente una serie de suposiciones sobre el tipo paramétrico, en este caso, T.

Por ejemplo, la forma en que hemos declarado el constructor implica que T ha de poseer públicamente un constructor por omisión[19], *T()*. También es necesario que posea destructor, constructor de copia y operador de asignación públicos (si queremos realizar las operaciones usuales). Asimismo se exige que posea *operator* `+=`, *operator* `-=` y *operator* `*=` para poder realizar sus contrapartidas vectoriales y que sus valores se puedan mostrar mediante una sobrecarga apropiada del operador de inserción.

[19]Se dice entonces que T es «constructible por omisión». Por eso los tipos fundamentales se comportan *como si* tuvieran constructor por omisión (por ejemplo, *int()*, *double()*, etc.).

Los tipos numéricos cumplen sobradamente todos estos requisitos. Y lo que es más importante: también lo hará cualquier *Vector* de un tipo T que a su vez los cumpla.

A continuación se define la clase *Matriz* a partir de un «*Vector* de *Vector* de tipo *T*».

matriz.h

```
1  #ifndef MATRIZ_H_
2  #define MATRIZ_H_
3
4  #include "vector.h"
5  #include <cassert>
6
7  template <typename T> class Matriz {
8  public:
9    explicit Matriz(size_t m = 1, size_t n = 1);
10   Matriz(const Vector<T>& v);
11   Vector<T>& operator [](size_t i);
12   const Vector<T>& operator [](size_t i) const;
13   Matriz& operator +=(const Matriz& a);
14   Matriz& operator -=(const Matriz& a);
15   Matriz& operator *=(const Matriz& a);
16   Matriz& operator *=(const T& k);
17   size_t filas() const;
18   size_t columnas() const;
19   void mostrar() const;
20 protected:
21   size_t m, n;                  // dimensión
22   Vector<Vector<T> > a;         // elementos
23 };
24
25 // Constructores
26
27 template <typename T> inline
28 Matriz<T>::Matriz(size_t m, size_t n):
29   m(m), n(n), a(m, Vector<T>(n)) {}
30
31 // Constructor de conversión
32
33 template <typename T> inline
34 Matriz<T>::Matriz(const Vector<T>& v):
35   m(1), n(v.dimension()), a(m, v) {}
36
37 // Dimensión
38
39 template <typename T> inline
40 size_t Matriz<T>::filas() const { return m; }
```

```
41
42  template <typename T> inline
43  size_t Matriz<T>::columnas() const { return n; }
44
45  // Operadores de índice
46
47  template <typename T> inline
48  Vector<T>& Matriz<T>::operator [](size_t i)
49  {
50    assert(i < m);
51    return a[i];
52  }
53
54  template <typename T> inline
55  const Vector<T>& Matriz<T>::operator [](size_t i) const
56  {
57    assert(i < m);
58    return a[i];
59  }
60
61  // Operador de suma
62
63  template <typename T> inline
64  Matriz<T> operator +(const Matriz<T>& a, const Matriz<T>& b)
65  {
66    assert(a.filas() == b.filas() && a.columnas() == b.columnas());
67    return Matriz<T>(a) += b;
68  }
69
70  // Operador de resta
71
72  template <typename T> inline
73  Matriz<T> operator -(const Matriz<T>& a, const Matriz<T>& b)
74  {
75    assert(a.filas() == b.filas() && a.columnas() == b.columnas());
76    return Matriz<T>(a) -= b;
77  }
78
79  // Operador de auto-producto
80
81  template <typename T> inline
82  Matriz<T>& Matriz<T>::operator *=(const Matriz<T>& a)
83  {
84    assert(columnas() == a.filas());
85    return *this = *this * a;
86  }
87
88  // Operadores de producto externo
89
```

```
90  template <typename T> inline
91  Matriz<T> operator *(const Matriz<T>& a, const T& k)
92  {
93    return Matriz<T>(a) *= k;
94  }
95
96  template <typename T> inline
97  Matriz<T> operator *(const T& k, const Matriz<T>& a)
98  {
99    return a * k;
100 }
101
102 // Definiciones de plantillas no «inline»
103
104 #include "matriz.cpp"
105
106 #endif
```

El primer constructor (que en realidad, es triple) no permite especificar un valor inicial para los elementos de la matriz: éstos se inicializarán de acuerdo a lo especificado en la clase *Vector*<*T*>, es decir, con el constructor por omisión de *T*.

Mención especial merecen los operadores de índice: devuelven referencias a *Vector*; de este modo, si *a* es de tipo *Matriz*<*T*>, las siguientes expresiones son equivalentes:

```
a[i][j]
(a.operator [](i)).operator [](j)
(a.Matriz<T>::operator [](i)).Vector<T>::operator [](j)
```

El primer *operator* `[]` es de la clase *Matriz* y devuelve una referencia a un objeto de la clase *Vector*, sobre el que se aplica el segundo. Así, la comprobación de rango se realiza sobre ambos índices.

matriz.cpp

```
1  #include "matriz.h"
2  #include <iostream>
3  using namespace std;
4
5  // Operador de auto-suma
6
7  template <typename T>
8  Matriz<T>& Matriz<T>::operator +=(const Matriz<T>& a)
9  {
```

```
10    assert(filas() == a.filas() && columnas() == a.columnas());
11    for (size_t i = 0; i < filas(); ++i)
12      (*this)[i] += a[i];
13    return *this;
14  }
15
16  // Operador de auto-resta
17
18  template <typename T>
19  Matriz<T>& Matriz<T>::operator -=(const Matriz<T>& a)
20  {
21    assert(filas() == a.filas() && columnas() == a.columnas());
22    for (size_t i = 0; i < filas(); ++i)
23      (*this)[i] -= a[i];
24    return *this;
25  }
26
27  // Operador de producto
28
29  template <typename T>
30  Matriz<T> operator *(const Matriz<T>& a, const Matriz<T>& b)
31  {
32    assert(a.columnas() == b.filas());
33    Matriz<T> c(a.filas(), b.columnas());
34    for (size_t i = 0; i < a.filas(); ++i)
35      for (size_t j = 0; j < b.columnas(); ++j)
36        for (size_t k = 0; k < a.columnas(); ++k)
37          c[i][j] += a[i][k] * b[k][j];
38    return c;
39  }
40
41  // Operador de auto-producto externo
42
43  template <typename T>
44  Matriz<T>& Matriz<T>::operator *=(const T& k)
45  {
46    for (size_t i = 0; i < filas(); ++i)
47      (*this)[i] *= k;
48    return *this;
49  }
50
51  // Mostrar
52
53  template <typename T>
54  void Matriz<T>::mostrar() const
55  {
56    for (size_t i = 0; i < filas(); ++i)
57      (*this)[i].mostrar();
58  }
```

Básicamente, se exige sobre T lo mismo que se le exigió para *Vector*, si bien el producto matricial hace que ahora sea necesario también un *operator* *.

Para finalizar el ejemplo, veamos un posible programa de prueba. En él se especializa *Matriz* para obtener *Matriz<double>*. Esto implica, por la definición de *Matriz*, una doble especialización de *Vector* (y de *vector*, claro está); por un lado, se crea el tipo *Vector<double>* y por otro, el tipo *Vector<Vector<double> >*.

Pero no sólo se especializan los tipos. El programa emplea los operadores de producto externo por la izquierda y de producto matricial; el primero se reduce a su versión por la derecha. Así, todos estos operadores externos necesitan ser especializados.

prueba.cpp

```
1  #include "matriz.h"
2  #include <iostream>
3  #include <algorithm>
4  using namespace std;
5
6  Matriz<double> identidad(size_t m = 1, size_t n = 1);
7
8  int main()
9  {
10   Matriz<double> a = 2.0 * identidad(4, 2);
11   Matriz<double> b = 3.0 * identidad(2, 4);
12   Matriz<double> c = a * b;
13
14   cout << "a = " << endl;
15   a.mostrar();
16   cout << "b = " << endl;
17   b.mostrar();
18   cout << "c = a * b" << endl;
19   c.mostrar();
20 }
21
22 // Matriz identidad
23
24 Matriz<double> identidad(size_t m, size_t n)
25 {
26   Matriz<double> c(m, n); // matriz nula
27   for (size_t i = 0; i < min(m, n); ++i)
28     c[i][i] = 1.0;
29   return c;
30 }
```

Empleo de `friend`

Las clases paramétricas, al igual que las normales, pueden tener clases y funciones amigas. Una función amiga que no utilice plantillas es *universal* a todas las especializaciones de la clase paramétrica. Una función amiga que emplee plantillas es *específica* de su clase especializada.

EJEMPLO:

En el siguiente fragmento, la función *auxiliar()* es universal a todas las especializaciones de la plantilla. Sin embargo, la sobrecarga de *operator* `*` es específica de cada especialización (existe una por cada T empleado).

```
template <typename T> class Matriz {
public:
  // ...
  friend void auxiliar();
  friend Vector<T> operator *(const Matriz<T>&, const Vector<T>&);
  // ...
};
```

Miembros estáticos

Una clase paramétrica puede poseer miembros estáticos, siendo éstos específicos de cada especialización, nunca universales.

EJEMPLO:

Supongamos la siguiente declaración:

```
template <typename T> class C {
public:
  static int n; // específico
  // ...
};
```

Si ahora se definen dos objetos:

```
C<int> v1;
C<double> v2;
```

las variables estáticas *C<int>::n* y *C<double>::n* son distintas.

4.8.2. Parámetros de plantilla

Una plantilla puede estar compuesta por varios parámetros. Así, el siguiente operador utiliza dos parámetros de tipo para permitir comparar vectores que no sean necesariamente del mismo tipo base, sino de tipos «compatibles».

```
template <typename T1, typename T2>
bool operator ==(const vector<T1>& a, const vector<T2>& b)
{
  const size_t n = a.size();

  if (n != b.size())
    return false;
  for (size_t i = 0; i < n; ++i)
    if (a[i] != b[i])
      return false;
  return true;
}
```

Los parámetros formales de una plantilla no se reducen exclusivamente a parámetros de tipo, pueden especificarse también parámetros de un tipo preexistente (por ejemplo, un entero o un parámetro de tipo que haya aparecido previamente en la plantilla).

Los parámetros reales de una plantilla han de ser en todo caso conocidos en tiempo de compilación.

Ejemplo:

La siguiente clase representa un búfer finito de manera muy eficiente, empleando un vector de bajo nivel de elementos de un tipo arbitrario. El búfer también tendrá un tamaño arbitrario definido en tiempo de compilación mediante un parámetro de plantilla.

```
template <typename T, size_t n> class Buffer {
  T b[n];
  // ...
};

// ...

V<char, 20> a, b;
V<char, 10> c;

a = b;    // bien
c = a;    // ERROR, se detecta en tiempo de compilación
```

4.8.3. Omisión de parámetros de plantilla

Al declarar un parámetro de plantilla se puede proporcionar un valor por omisión para él, de manera similar a lo que ocurre con los parámetros normales.

Ejemplo:

Podemos hacer que, por omisión, el tipo de los elementos almacenados en un búfer sea *char*, y el número máximo de elementos, 256.

```
template <typename T = char, size_t n = 256> class Buffer {
  T b[n];
  // ...
};

// ...

V<double> a;
V<> b;
```

Así, *a* será un búfer de, a lo sumo, 256 elementos de tipo *double*. Nótese cómo al definir *b* se omiten ambos parámetros de plantilla.

4.8.4. Exportación

Hay dos enfoques a la hora de organizar el código fuente cuando se emplean plantillas:

1. Definir las plantillas en la misma unidad de traducción en la que se emplean (previamente a su uso).
2. Declarar las plantillas en la misma unidad de traducción en la que se emplean (previamente a su uso) y definirlas posteriormente o en otra unidad de traducción.

El primer enfoque no presenta problemas; además corresponde al compilador generar código únicamente cuando se necesita. Así, las funciones paramétricas se tratan como las funciones «en línea», siendo por lo tanto susceptibles de inclusión en ficheros de cabecera. Es apropiada sobre todo en programas pequeños o en los que las plantillas se emplean en un único fichero.

El segundo enfoque corresponde a la idea de compilación separada, que permite mantener independientes las declaraciones de las definiciones. Es, sin duda, la mejor solución general.

Sin embargo, existen problemas técnicos para su empleo: hay que hacer accesible la definición a las restantes unidades de traducción, ¿cómo sabría si no el compilador qué especializaciones generar al compilar un fichero con definiciones paramétricas que se emplean en otro?

Para lograr que una definición paramétrica sea accesible en una unidad de traducción distinta es necesario anteponer a ella la palabra reservada `export`[20]. A esto se le conoce como «exportar» la plantilla.

[20]Desgraciadamente, esto aún no está disponible en muchos compiladores, ya que suele implicar cambios importantes en el proceso de enlazado.

Ejercicios

E4.1. Defina una clase para trabajar con fechas que contenga:

- Una función miembro que permita crear una fecha a partir del día, el mes y el año. Para simplificar, suponga que la fecha especificada es correcta.
- Una función miembro observadora que muestre la fecha. Deberá presentar los meses empleando su nombre.
- Tres miembros de datos: el día, el mes y el año.

Suponiendo, como hemos hecho, que la función de creación siempre produce una fecha correcta, comente las implicaciones que tendría permitir que los miembros de datos fueran públicos pudiendo así ser modificados desde el exterior.

E4.2. En la clase del ejercicio anterior sobrecargue la función miembro que crea una fecha, de manera que si no recibe parámetros obtenga la información del sistema. Escriba un pequeño programa de prueba.

PISTA: Emplee las funciones *time()* y *localtime()* de C cuyas declaraciones se encuentran disponibles en `<ctime>`. La *struct tm* cuya dirección devuelve esta última función contiene en sus campos *tm_mday*, *tm_mon* y *tm_year* los valores necesarios (*tm_year* es el número de años transcurridos desde 1900).

E4.3. ¿Qué ocurre si una función miembro no declarada como `const` se aplica a un objeto `const`? ¿Y a una referencia `const` de un objeto que no sea `const`?

E4.4. Las protecciones de acceso a los miembros de una clase «se pierden» cuando el programa está compilado y en ejecución. Para probarlo, intente acceder a los miembros privados de la siguiente clase, cambiando sus valores directamente.

```
// Acceso malintencionado a miembros privados

#include <iostream>
using namespace std;

class X {
  int i;
  char c;
public:
  X(int ii, char cc) { i = ii; c = cc; }
  friend ostream& operator <<(ostream& s, const X& x)
    {
```

```
      return s << x.c << ", " << x.i;
    }
};

int main()
{
  X x(65, 'A');
  cout << x << endl;
  // cambio de los miembros privados de forma truculenta
  /* ... rellene Ud. esto;
     haga que X::i valga 400 y X::c valga 'x', por ejemplo
     ahora comprobamos los nuevos valores
  */
  cout << x << endl;
}
```

¿Cree que eso está bien? ¿Cómo es que el compilador de C++ lo permite?

PISTA: Haga uso del operador `reinterpret_cast` y de punteros. Piense en la representación en memoria de la clase. Pruebe si funciona suponiendo que todos los miembros de datos se almacenan contiguos, sin huecos.

E4.5. ¿Qué cree que pasa en el siguiente código en cuanto a las definiciones de *x1*, *x2* y *x3*? Piense antes de compilar y probar. ¿Dará errores de compilación? ¿Dónde y por qué? Después escriba la función y compílela para comprobar su teoría.

```
1  // Saltos sobre constructores
2
3  class X {
4  public:
5    X() {}
6  };
7
8  void f(int i)
9  {
10   if (i < 10)
11     goto salto;
12   X x1;         // llamada al constructor
13  salto:
14   switch (i) {
15   case 1:
16     X x2;       // llamada al constructor
17     break;
18   case 2:
19     X x3;       // llamada al constructor
20     break;
```

```
21    }
22  }
```

E4.6. Diseñe la clase *Fecha* para almacenar una fecha compuesta de día, mes y año. Deberá tener un constructor predeterminado y uno que acepte tres parámetros: día, mes y año en este orden. El constructor predeterminado creará el objeto con la fecha de *hoy*. Otro constructor aceptará una cadena de caracteres de bajo nivel que contendrá una fecha en el formato que Ud. elija. Tendrá que comprobar que las fechas sean correctas. Si no lo fueran, como aún no se han visto las excepciones, emplee la macro *assert()* definida en `<cassert>` (consulte el Manual o el libro de la Biblioteca Estándar de C).

Sobrecargará los operadores de incremento y decremento, que modificarán la fecha al día de *mañana* y *ayer* respectivamente. También definirá los operadores de suma y resta de fecha y entero, que incrementarán o decrementarán la fecha el número de días especificado por el segundo operando; y los operadores de suma y resta con asignación.

Definirá tres métodos observadores que devolverán los tres atributos: día, mes y año.

Como siempre, escriba un pequeño programa para probar la clase, y antes de nada, el *Makefile*.

E4.7. Suponga que no quisiéramos que las fechas se pudieran copiar unas a otras. ¿Cómo haría para que el compilador diera un error ante un intento de pasar una fecha por valor a una función por ejemplo, o asignar una fecha a otra?

E4.8. Modifique la clase *NombreCaracter* del fichero `nombrecaracter.h` (§4.3.5) de forma que el compilador dé error en la línea

```
c = i;
```

del programa en el fichero `latin1.cpp`; pero de todas formas, la conversión podrá tener lugar si la ponemos explícita. Compile el programa para ver el error que le da el compilador; haga luego la conversión explícita y recompile: no debe dar error ahora.

E4.9. Diseñe una clase *Cadena* para almacenar cadenas de caracteres como una pobre imitación de *string*. Los atributos serán un puntero a caracteres y un tamaño. Los constructores serán:

- el predeterminado, que construirá una cadena vacía.
- uno que reciba un entero n y un carácter, prefijado al espacio en blanco. Creará una cadena de n repeticiones de dicho carácter.

Este constructor *no* podrá usarse como de conversión de entero a *Cadena*.

- uno de conversión de cadena de caracteres de bajo nivel a *Cadena*.
- el de copia.

Sobrecargará los siguientes operadores:

- el de asignación, si es preciso, y el de asignación con conversión desde una cadena de caracteres de bajo nivel.
- el de conversión de *Cadena* a cadena de bajo nivel.
- el de suma, que concatenará dos *Cadena*s.
- el de suma y asignación, que añade una *Cadena* al final.
- el de índice, que devolverá el carácter correspondiente. Compruebe que el índice esté en el rango correcto; haga uso de la macro *assert()* como en un ejercicio anterior.

Complete la clase con el destructor, si es preciso, y con un método observador que devuelva la longitud de la cadena. Como extensión, puede añadir otros métodos que se le ocurran (subcadenas, búsqueda de caracteres o subcadenas, etc.). Por último, haga un programa de prueba; y antes de nada, el *Makefile*, como siempre.

E4.10. Implemente la asociación *capital de* entre la clase *Ciudad* y la clase *Pais* de la figura 4.2.

E4.11. En el ejemplo de §4.6.2 falta el fichero `lista.cpp` que se muestra a continuación:

lista.cpp

```
1  #include <iostream>
2  #include <cassert>
3  #include <algorithm>
4  #include "lista.h"
5  using namespace std;
6
7  bool Lista::vacia() const
8  {
9    return l.empty();
10 }
11
12 int Lista::primero() const
13 {
14   assert(!vacia());
15   return l.front();
16 }
```

```
17
18  int Lista::ultimo() const
19  {
20    assert(!vacia());
21    return l.back();
22  }
23
24  void Lista::insertarPrincipio(int e)
25  {
26    l.push_front(e);
27  }
28
29  void Lista::insertarFinal(int e)
30  {
31    l.push_back(e);
32  }
33
34  void Lista::eliminarPrimero()
35  {
36    assert(!vacia());
37    l.pop_front();
38  }
39
40  void Lista::eliminarUltimo()
41  {
42    assert(!vacia());
43    l.pop_back();
44  }
45
46  void Lista::mostrar() const
47  {
48    copy(l.begin(), l.end(), ostream_iterator<int>(cout, " "));
49    cout << endl;
50  }
```

Intente entenderlo buscando información sobre la clase *deque*. Compile y pruebe el programa.

E4.12. Reescriba la clase *Pila* del ejercicio anterior utilizando generalización, en concreto herencia pública, en lugar de agregación.

E4.13. ¿Cuál de las dos formas anteriores de implementar la clase *Pila* considera menos adecuada? ¿Por qué? ¿Qué pequeña modificación habría que hacer para paliar los inconvenientes?

E4.14. Modifique las clases *Estudiante* y *Doctorando* de §4.6.3 utilizando polimorfismo y funciones miembro virtuales, para que se seleccione el método adecuado en tiempo de ejecución entre los de la clase base o derivada y para que se solucione el problema planteado en §4.7.

E4.15. Implemente una clase *Persona* que tenga los atributos: *nombre*, *dni*, *telefono* y *direccion*; y los métodos: *mostrar()*, *cambiarTelefono()* y *cambiarDireccion()*. Especialícela en dos, *Alumno* y *Profesor*, añadiendo los atributos y operaciones que considere adecuados para cada una de las clases.

A continuación escriba una clase *Asignatura* que tenga los atributos: *nombre*, *codigo*, *temario* y *area*; y los métodos: *mostrar()* y *cambiarTemario()*.

Por último implemente dos asociaciones bidireccionales *varios a varios*, una llamada *matriculado* entre *Alumno* y *Asignatura*; y otra llamada *imparte* entre *Profesor* y *Asignatura*.

Haga un programa de prueba que muestre los alumnos matriculados y los profesores que imparten clases en cada asignatura.

¿Sería lo mismo establecer una única asociación bidireccional *varios a varios* entre la clase *Persona* y *Asignatura* en lugar de las dos propuestas anteriormente?

E4.16. Un polinomio $\sum_{i=1}^{n} a_i \cdot x^i \in K[x]$ se puede representar mediante un vector $[a_0, \ldots, a_n] \in K^n$. Programe una función, *evaluar()*, que evalúe un polinomio en un punto a (es decir, que calcule $\sum_{i=1}^{n} a_i \cdot a^i$). La función dependerá de un tipo K, recibirá un *vector*<K> y un valor de tipo K y devolverá como resultado un valor de dicho tipo.

Escriba también un pequeño programa de prueba que evalúe el polinomio $x^2 + 2x + 1$ en los 10 primeros números naturales (emplee la especialización a *int* de la función).

E4.17. ¿Funcionaría directamente su función *evaluar()* del ejercicio anterior con tipos más complejos? Por ejemplo, ¿lo haría con polinomios de matrices?

Pista: Determine qué requisitos impone su definición de *evaluar()* sobre el tipo K.

E4.18. Escriba una pila paramétrica empleando un vector de bajo nivel de longitud prefijada. Emplee únicamente un parámetro de tipo para la plantilla. La clase se llamará *Pila* y dispondrá públicamente de las siguientes funciones miembro:

- *Pila()*, que construirá la pila vacía.
- *vacia()*, que dirá si una pila está o no vacía.
- *cima()*, que devolverá el elemento de la cima de la pila.
- *apila()*, que colocará un elemento en la cima de la pila.
- *desapila()*, que eliminará la cima de la pila sin devolver nada.

Se emplearán asertos para evitar situaciones indeseadas como las que se producirían al apilar en una pila llena y al hallar la cima, o desapilar, de una pila vacía.

E4.19. ¿Qué tendría que hacer para que la longitud máxima de la pila del ejercicio anterior formara parte del tipo de ésta? ¿Qué implicaciones tendría esto en la práctica?

E4.20. Cree una clase paramétrica *Polinomio* para representar elementos de $K[x]$ en la forma indicada en el ejercicio **E4.16**. Incluya las operaciones habituales y sobrecargue *operator* `()` para evaluar polinomios.

Capítulo 5

Excepciones

5.1. Introducción

Supongamos un pequeño (relativamente) programa no interactivo, como uno de manejo de ficheros de un sistema operativo como DOS (o derivados) o Unix; pongamos *mkdir*, que sirve para crear un directorio. Sigamos suponiendo que damos la orden pero sin el parámetro correspondiente, o no tenemos permiso para crear un directorio. ¿Qué debería hacer el programa? Claramente, indicárnoslo mediante el oportuno mensaje de error y acabar inmediatamente, pues no es un programa interactivo.

Supongamos ahora un programa interactivo más complejo, como un editor o procesador de textos de un paquete ofimático. Pulsamos el botón de abrir un fichero y en el cuadro de diálogo escribimos un nombre de fichero incorrecto, o que por algún motivo no podemos abrir. ¿Sería lógico que el programa acabara con un mensaje de error? Más bien debería, sí, informarnos de él, pero dándonos la oportunidad de seguir intentándolo o cancelar la operación.

Bajando de nivel, supongamos que escribimos un programa que hace uso de una biblioteca hecha por otras personas; no disponemos del código fuente, sólo del archivo con los módulos objeto y los ficheros de cabecera. Si llamamos a una función de esa biblioteca pasándole un parámetro con un valor incorrecto, ¿sería lógico que, en tiempo de ejecución, el programa se acabara dando un diagnóstico en la salida de errores? Nuestro programa puede ser gráfico, y esa salida de errores estar conectada a la consola, que no veríamos, o a un fichero de registro, que tampoco veríamos inmediatamente. Quizá nuestro programa sea demasiado «importante» como para acabar en ese punto sin posibilidad de volver atrás a corregir el error.

Todo esto indica que hay diversas formas de tratar los errores, unas más adecuadas que otras según el caso.

El autor de una biblioteca, que en C++ seguramente no se limita a un conjunto de funciones, sino de clases, cada una con sus correspondientes métodos, probablemente se topará con diversos errores que pueda cometer el usuario, sin saber qué hacer con ellos. Las *excepciones*, que vemos en este capítulo, son un mecanismo de control que permite manejar este tipo de errores o problemas de forma que cuando en una función (externa, o interna a una clase), normalmente de una biblioteca, se prevea el encuentro con uno de ellos, se detecte pero se deje el manejo del error en manos del usuario, del *cliente* de la biblioteca o función.

Cuando se detecta un error, el autor de la función *lanza* una excepción. El cliente, si quiere, puede *capturarla* y obrar como le plazca.

5.2. Lanzamiento de excepciones

Antes de entrar en materia, veamos las alternativas que tiene el autor de una biblioteca en ausencia del mecanismo de excepciones.

Cuando éste detecta un problema que no puede tratar directamente o no sabe cómo, la función o método puede optar por:

1. terminar el programa.

   Esto es lo que ocurre incluso en el caso de excepciones cuando el cliente no las captura. A veces, esto es lo natural, como en el primer ejemplo mencionado en §5.1, pero en otros casos hay cosas mejores que se pueden hacer. El autor de una biblioteca normalmente no sabe nada acerca del propósito y estrategia general del programa que la usa, y puede no ser admisible emplear *exit()* y aún menos *abort()* para terminar el programa.

   Sin embargo, ésta es la alternativa que hemos empleado hasta ahora en nuestros ejercicios y ejemplos mediante el empleo de la macro *assert()*, que prueba una condición o expresión y, si es falsa (0), presenta un diagnóstico en la salida de errores y llama a *abort()* para terminar el programa abruptamente. Esta técnica puede ser admisible en la fase de depuración de un programa, y, afortunadamente, todas las aserciones pueden anularse sin tener que borrarlas una a una recompilando el programa con la macro *NDEBUG* definida, se supone que una vez comprobado el funcionamiento correcto. Pero a veces éste está determinado por datos que el programa adquiere en tiempo de ejecución, con lo que no vale simplemente anular las aserciones, ni dejarlas tampoco.

2. devolver un valor que represente «error».

Ésta es la alternativa empleada por muchas funciones de la biblioteca estándar de C, donde no existe el mecanismo de excepciones, y por otras muchas bibliotecas. Esta técnica no vale cuando cualquier valor devuelto por la función pueda ser un valor correcto, evidentemente. Y aunque sí se pueda devolver un valor con el significado claro de «error» sin que interfiera con otro válido, el cliente tendría que estar, en cada llamada a la función, comprobando el valor devuelto, lo que hace el código más pesado de leer, y aumenta su tamaño, además de que puede que el cliente se olvide de comprobar el valor devuelto.

3. devolver un valor legal y dejar el programa en un estado ilegal.

   Esta alternativa también es empleada donde no se puede la anterior en algunas funciones de la biblioteca estándar de C, sobre todo funciones matemáticas, y, a veces en conjunción con la anterior, en muchas funciones de la biblioteca POSIX.1. Dichas funciones dejan un valor en una variable global llamada *errno*, definida en la biblioteca estándar de C, que indica el error producido. Pero el cliente puede olvidarse de comprobar el estado de esta variable; y el empleo de variables globales es delicado en presencia de concurrencia.

4. llamar a una función que pueda suministrar el cliente en caso de «error».

   Esta alternativa es quizás la mejor de todas, pero en ausencia del mecanismo de excepciones, la función se encuentra en uno de los casos anteriores, si bien al menos es el cliente el que la define.

5.2.1. Excepciones

¿Qué es exactamente una «excepción»? Bajo el punto de vista del compilador es un objeto; es decir, un valor, de un tipo fundamental incorporado en el lenguaje o de un tipo definido por el usuario (normalmente una clase); un objeto que se *lanza* (en inglés, *throw*) en una función donde se detecta un error o problema, y que se puede recoger o capturar (en inglés, *catch*) en otra donde se pueda tratar o procesar (en inglés, *try*) el suceso. Si no se capturara, está previsto un comportamiento bien definido que veremos más adelante.

5.2.2. La instrucción `throw`

Una excepción se lanza mediante una instrucción `throw` con la sintaxis siguiente:

`throw` [*expresión*];

La forma donde se omite la *expresión* se emplea para re-lanzar una excepción a un nivel superior. Estudiaremos este caso más adelante, en §5.3.2.

Primeramente, con la *expresión* se crea un objeto del tipo correspondiente (y por tanto se llama a su constructor si se trata de un tipo definido por el usuario); ese objeto no existe durante una ejecución «normal» del programa: sólo se creará si se lanza la excepción.

Ejemplo:

```
class Error { /* ... */ };

void f() {
  ...
  Error e; // constructor predeterminado
  ...
  throw e; /* se lanza un objeto temporal, copia de «e»,
            * creado con el constructor de copia, y se
            * destruye «e» con el destructor */
  ...
  throw Error(); /* se crea un objeto temporal con el ctor.
                  * predeterminado, se copia en otro temporal,
                  * que es el que se lanza, con su constructor
                  * de copia, y se destruye el primer temporal
                  */
  ...
  throw new Error; /* se lanza un puntero a Error, que apunta
                    * a un objeto anónimo creado dinámicamente
                    */
  ...
}
```

Normalmente los objetos que se lanzan son de una clase especial que se creará el programador para representar los errores, por lo que la primera forma no se debe emplear: no tiene sentido crear un objeto de tipo *Error* si no se va a producir ninguno.

Entonces la función acaba y devuelve por valor dicho objeto como si de un `return` se tratara, aunque el tipo de devolución de ésta sea otro distinto, o *void*, o incluso ni siquiera éste, como pasa en los constructores.

Sin embargo, el control no pasa en este caso al punto de llamada a la función, como ocurre con un `return` normal, sino a un manejador de excepción apropiado, si lo hay, que puede estar bastante *lejos* de ese punto de llamada.

Además, los objetos automáticos que se hubieran creado con éxito hasta el lanzamiento de la excepción (y sólo ellos) se destruyen, llamando a sus des-

tructores, puesto que la función termina. El objeto lanzado se destruye sólo cuando se acaba el manejador correspondiente (los manejadores se explican en §5.3.2).

¿Y qué objeto lanzar? Podría lanzarse un simple número que significara un código de error, o una cadena de caracteres con un mensaje. No obstante, es mucho más recomendable crear una clase o tipo para cada error diferente. Algunos errores pueden estar relacionados entre ellos, y así deberían estar también las clases que diseñáramos para representarlos. La idea es encapsular la información que debe llevar la excepción en una clase, incluyendo su nombre, para que el usuario que vaya a tratar con ellas sepa su significado. Si no hace falta incluir ninguna información que haya que pasar al manejador, basta con una clase vacía[1] con un nombre apropiado.

Ejemplo:

La siguiente clase representa un número de identificación fiscal, o NIF, que comprende el número del documento nacional de identidad o DNI más una letra de control calculada en función de dicho número.

```
class Nif {
  unsigned int dni;
  char letra;
  bool letra_valida();
public:
  // Clase de excepción, anidada
  class LetraInvalida { /* vacía */ };
  // Constructor
  Nif(unsigned int n, char ltr): dni(n), letra(ltr)
  {
    if (not letra_valida()) throw LetraInvalida();
  }
  // ...
};
```

Se ha creado una clase llamada *LetraInvalida* para representar el error consistente en que la letra suministrada al constructor no sea la correcta. Se considera que la información que lleva el propio nombre de la clase es suficiente, por lo que se define vacía. Se ha creado dentro de la propia clase *Nif*, pues no tiene sentido fuera de ella: pertenece a esta clase (hay que acordarse, cuando se haga referencia a ella fuera de la clase, de utilizar el operador de resolución de ámbito).

[1] En C++, a diferencia de C, puede haber `structs` y clases sólo con funciones miembro, o incluso vacías.

No hace falta definir antes un objeto del tipo de la excepción y lanzarlo: tras `throw` hacemos una llamada al constructor (en el ejemplo, al no haberse definido ninguno, al predeterminado proporcionado automáticamente por el compilador, que no hace nada) para crear un objeto temporal anónimo que se copia en otro que es el que se lanza, y el primer temporal se destruye inmediatamente. Esta técnica es muy común: sólo debería crearse el objeto excepción cuando se produjera una. Las excepciones se supone que son (como parece evidente del nombre) excepcionales, y por tanto un programa con excepciones debería ejecutarse igual de rápido que otro sin ellas en el caso de que no se produzca ninguna.

Cuando se lanza una excepción en un constructor, el objeto no ha terminado de construirse, y no se llamará a su destructor. En el ejemplo anterior, esto no tiene ninguna importancia al ser los atributos simples números, pero sí empieza a tenerla cuando en el constructor se adquieren recursos que luego deban liberarse con el destructor.

5.2.3. La lista `throw`

Uno no tiene por qué informar a la persona que emplee nuestras clases o funciones de qué excepciones puedan lanzar, pero esto sería muy poco considerado porque significaría que dicha persona no podría estar segura de cómo capturar todas las posibles excepciones. Por supuesto, si tiene el código fuente, puede mirarlo buscando todas las instrucciones `throw`, pero muy a menudo una biblioteca se entrega en un archivo, ya compilada, sin fuentes. Por ello, C++ proporciona un mecanismo por el cual se puede informar educadamente al cliente de qué excepciones exactamente podrá lanzar una función o método, de forma que así éste podrá escribir los manejadores apropiados cuando las capture. Este mecanismo se conoce como *especificación de excepciones* o *lista throw*, y forma parte de la declaración de la función, apareciendo tras la lista de parámetros.

Se reutiliza la palabra reservada `throw`, seguida de una lista de excepciones separadas por comas, entre paréntesis. La lista vacía significa que la función no lanzará ninguna excepción.

Ejemplo:

Siguiendo con la clase *Nif*, el constructor se escribiría mejor así:

```
Nif(unsigned int n, char ltr) throw(LetraInvalida)
: dni(n), letra(ltr)
{
```

```
  if (not letra_valida()) throw LetraInvalida();
}
```

Si la lista `throw` vacía significa que la función no va a lanzar ninguna excepción, la falta de la especificación de excepciones significa que la función puede lanzar cualquier excepción, o ninguna.

Ejemplo:

```
void f() throw(MuyGrande, MuyChico, DivCero);
void g();
void h() throw();
```

Las funciones anteriores no reciben ningún parámetro ni devuelven nada. La primera, *f()*, puede lanzar excepciones de los tres tipos especificados; *g()* podría lanzar cualquier excepción, o no lanzar ninguna, y *h()* no podrá lanzar ninguna excepción. Con esta información, el cliente sabe que cuando llame a *f()* debería capturar esos tres tipos de excepciones, y que cuando llame a *h()* no hará falta que se preocupe de capturar nada. En cambio, al llamar a *g()* no sabrá qué hacer a este respecto.

Es recomendable siempre, cuando se escribe un programa o biblioteca con excepciones, escribir la lista `throw` para cada función. Por seguir un buen estilo, para documentación, y para facilitarle la vida al que emplee nuestras funciones.

> Se podría pensar que las especificaciones de excepciones no están muy bien diseñadas, de forma que el ejemplo anterior debería ser mejor:
>
> ```
> void f() throw(MuyGrande, MuyChico, DivCero);
> void g() throw(...); // cualquier excepción
> void h();            // ninguna excepción
> ```
>
> Esto no es lo mejor por varios motivos. Uno, que no siempre se sabe con certeza qué excepción puede lanzar una función, porque aunque en ella no aparezca un `throw`, puede lanzar una excepción porque llame a otra función que sí la lance y no se recoja en ésta, o porque un operador de C++ lance una excepción estándar. Y, lo principal, para mantener inalterado el código escrito antes de que se inventara el mecanismo de excepciones, pues funciones antiguas pueden lanzar de pronto excepciones inadvertidamente si llaman a otras funciones que se hayan actualizado y ahora empleen este mecanismo.

5.3. Gestión de excepciones

Si una función lanza una excepción, se supone que se capturará y se tratará el error en otro sitio. Una de las ventajas del manejo de excepciones de C++ es que permite a uno concentrarse en el problema que realmente se está intentando resolver, mientras que los errores que ese código pueda producir se tratarán en otro sitio.

5.3.1. La instrucción `try`

Si en una función se lanza una excepción, o se llama a una función que lanza una excepción (y así recursivamente), esa función acabará en el proceso del lanzamiento. Si no se desea que `throw` provoque el abandono de una función, se puede establecer un bloque especial dentro de la función donde se está intentando resolver el problema real de programación y que potencialmente puede generar excepciones. Este bloque se llama el «bloque *try*», y es un bloque ordinario precedido de la palabra reservada `try`. Si comprende a la función entera, puede sustituir completamente al bloque del cuerpo de la función.

```
try {
  // código que puede generar excepciones ...
}
// ...
```

En ausencia del manejo de excepciones, un programador cuidadoso tendría que «rodear» cada llamada a función de código que comprobara si su valor devuelto fuera correcto y actuara en consecuencia, incluso si se llamara muchas veces a la misma función. Con el manejo de excepciones, todo se mete en un bloque *try* sin comprobar explícitamente los errores. Esto implica un código más claro y fácil de leer, porque su objetivo o tarea principal no está mezclado con la comprobación de errores.

5.3.2. La instrucción `catch`

Por supuesto, una excepción lanzada debe acabar en algún sitio, y éste es el *manejador de excepción*; habrá uno para cada tipo de excepción que se quiera capturar. Los manejadores de excepción se escriben inmediatamente tras el bloque `try`, y se nombran con la palabra reservada `catch`. Cada cláusula `catch` es como una función que no devuelve nada (ni siquiera *void*) y recibe un parámetro del tipo de la excepción que quiere capturar.

```
try {
  // código que puede lanzar excepciones
} catch(Tipo1 id1) {
   // trata con excepciones de Tipo1
} catch(Tipo2 id2) {
   // trata con excepciones de Tipo2
}
// etc...
```

Se pueden emplear los identificadores como parámetros formales de una función, y pueden ser objetos, punteros o referencias, según el caso; si no hacen falta porque no se van a emplear en el manejador, se pueden omitir; esto es frecuente por ejemplo cuando la excepción a capturar es una clase vacía cuyo nombre ya posee toda la información necesaria.

Ejemplo:

Sigamos con la clase *Nif*. La siguiente función pide al usuario un NIF y construye un objeto *Nif*; se emplea la técnica de la *rectificación* o *reanudación* mediante bucle, frente al otro modelo de manejo de excepciones: la *terminación*.

```
Nif lee_nif()
{
  for (;;) {
    cout << "Por favor, introduzca su número "
            "de DNI y su letra del NIF: ";
    unsigned int n;
    char c;
    cin >> n >> c;
    try {
      Nif nif(n, c); // posible excepción
      return nif;    // salida de la función
    } catch(LetraInvalida) {
      cerr << "Letra inválida. Por favor, repita.\a" << endl;
    } // se vuelve al bucle «infinito»
  }
}
```

Cuando dentro de un bloque `try` se lanza una excepción, el control se pasa inmediatamente al primer `catch` cuya *signatura* coincida con el tipo de la excepción. Se buscan de arriba abajo hasta el primero coincidente y, una vez ejecutado su código, se destruye el objeto lanzado y la excepción se considera manejada. La búsqueda acaba aquí; no como en las sentencias `switch` donde había que acabar cada `case` con un `break` si no se quería pasar al siguiente.

Una vez ejecutado el manejador apropiado, el control pasa a la instrucción posterior a todos los manejadores `catch`.

Esto implica que hay que ser cuidadoso con el orden en que se escriben los manejadores. Los más generales hay que ponerlos en los últimos lugares.

Ejemplo:

```
catch(void*) { }  // capturaría cualquier ptr., incluido char*
catch(char*) { }  // Mal, nunca se llamaría

catch(ErrorBase&) { }     // capturaría ErrorDerivado
catch(ErrorDerivado&) { } // Mal, nunca se llamaría
```

En el proceso de búsqueda del manejador apropiado, un objeto o referencia a un objeto de una clase derivada concordará con un manejador para la clase base (aunque si se lanza un objeto por valor en vez de por referencia, se perderá información en la copia al objeto de la clase base en el manejador). Si se lanza un puntero, se emplean las conversiones normales de punteros para buscar el manejador: así, un puntero a un objeto de una clase derivada, concordará con un manejador cuyo parámetro sea un puntero a un objeto de la clase base correspondiente.

Sin embargo, no se aplican conversiones automáticas definidas por el usuario, como puede verse en el siguiente programa:

```
1  /* Prueba de que no se aplican conversiones
2     definidas por el usuario en la búsqueda
3     de manejadores de excepciones.
4  */
5  #include <iostream>
6  using namespace std;
7
8  class Excepcion1 {};
9  class Excepcion2 {
10 public:
11   Excepcion2(Excepcion1&) { }   // ctor. de conversión
12 };
13
14 int main()
15   try
16 {
17   throw Excepcion1();
18 }
19 catch(Excepcion2) {
```

```
20    cout << "En catch(Excepcion2)" << endl;
21  }
22  catch(Excepcion1) {
23    cout << "En catch(Excepcion1)" << endl;
24  }
```

Aunque esté definida la conversión desde *Excepcion1* a *Excepcion2*, esta conversión no se efectúa durante el manejo de excepciones, y se acaba en el manejador de *Excepcion1*.

Captura de cualquier excepción

Como se ha explicado en §5.2.3, si una función no tiene lista `throw`, puede lanzar cualquier tipo de excepción. Una solución para no dejar una excepción sin capturar es definir un manejador que capture cualquier excepción. Esto se consigue escribiendo elipsis en la lista de parámetros (al estilo C):

Ejemplo:

```
catch(...) {
  cerr << "Se ha lanzado una excepción y la he capturado." << endl;
  // ...
}
```

Evidentemente, este manejador debe ser el último de la lista. Obsérvese que no tenemos ninguna información sobre el tipo de error producido ni podemos definir ningún parámetro formal donde copiar la excepción lanzada.

Relanzamiento de una excepción

A veces conviene relanzar la excepción capturada, por varios motivos:

- puede que en la función donde estemos sepamos o podamos tratar otros tipos de excepciones pero no éste.
- puede que sólo sepamos o podamos tratar parte del error, pero no todo.
- estamos en el manejador que captura cualquier excepción, el de los puntos suspensivos, y no tenemos por tanto información ninguna sobre la excepción.

En cualquiera de esos casos podemos relanzar la excepción producida con la instrucción `throw` sola:

EJEMPLO:

```
catch(...) {
  cerr << "Se ha lanzado una excepción." << endl;
  throw;
}
```

La excepción relanzada será capturada con suerte en un nivel superior, hasta llegar a *main()*.

5.3.3. La función *terminate()*

A estas alturas puede que se esté preguntando: ¿y qué pasa si no se captura una excepción en ningún sitio? Si ninguno de los manejadores `catch` que siguen a un bloque `try` concuerda con una excepción lanzada, ésta se pasa al siguiente contexto de mayor nivel; esto es, la función o bloque `try` donde se llamara al bloque que no ha capturado la excepción. Este proceso continúa hasta que en algún nivel un manejador capture la excepción: en este punto, ésta se considera «manejada» y no se sigue buscando.

Si ningún manejador captura la excepción en ningún nivel, siendo el superior la función *main()*, la excepción es «no capturada» o «no manejada». Este caso también se da cuando se lanza una nueva excepción antes de que la existente alcance a su manejador; el caso más común de esto ocurre cuando un constructor de un objeto excepción lanza a su vez otra.

Si una excepción no es capturada, se llama automáticamente a la función *terminate()*. Esta función está implementada como un puntero a función que no recibe ni devuelve nada. Si no se dice otra cosa, llama a la función *abort()*, de la biblioteca estándar de C, que acaba abruptamente la ejecución del programa; esto es, no se cierran ficheros abiertos, ni se vuelcan los *búferes*, ni se llama a los destructores de objetos globales o estáticos.

Se puede (y debe) cambiar este brusco comportamiento instalando una función escrita por el usuario que sea la que *terminate()* llame en lugar de *abort()*. Esta función se instala pasando su dirección a la función estándar *set_terminate()*, declarada en la cabecera estándar `<exception>`. La función *set_terminate()* devuelve un puntero a la función *terminate()* que se está reemplazando, por si se quiere restaurar luego.

De todas formas, aunque se haya instalado una función de terminación, no se llama a los destructores de los objetos globales ni estáticos, así que una alternativa mejor a dejar la responsabilidad a *terminate()* sería meter un bloque `try` en *main()* cuyo último capturador fuera el de la elipsis. Por lo general, una excepción no capturada debe considerarse un error de programación.

Nuestra función de terminación no debe recibir ningún parámetro ni devolver nada; además no debe regresar ni relanzar ninguna excepción, sino arreglar lo que se pueda y llamar a alguna función que termine el programa, como *exit()*, de la biblioteca estándar de C, declarada en `<cstdlib>`. Si se llama a *terminate()* se supone que es porque el problema no tiene recuperación posible.

Ejemplo:

En este ejemplo se muestra el empleo de *set_terminate()* y una excepción sin capturar.

terminator.cpp

```
/* Ejemplo de uso de set_terminate()
 * y de excepciones no capturadas
 */
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;

void terminator()
{
  cerr << "Sayonara, baby!" << endl;
  exit(EXIT_FAILURE);
}

void (*terminate_anterior)() = set_terminate(terminator);

class Chapuza {
public:
  class Fruta {};
  void f() {
    cout << "Chapuza::f()" << endl;
    throw Fruta();
  }
  ~Chapuza() { throw 'c'; }
};

int main()
  try
{
  Chapuza ch;
  ch.f();
} catch(...) {
```

```
33    cout << "En catch(...)" << endl;
34  }
```

La clase *Chapuza* no sólo lanza una excepción en su método *f()*, sino en su destructor. Como se ha dicho, ésta es una de las situaciones donde se produce una llamada a *terminate()*. Aunque parezca que se debería llegar al `catch` que captura todas las excepciones, se llama en su lugar a *terminate()* porque en el proceso de limpiar los objetos de la pila para manejar una excepción, se llama a otra en el destructor de *Chapuza*, y eso genera una segunda excepción, forzando la llamada a *terminate()*. Se deduce de esto que un destructor nunca debería lanzar una excepción ni provocar indirectamente el lanzamiento de una. Si se lanza una excepción en un destructor, debe tratarse allí mismo, metiendo en él un bloque `try` con los capturadores correspondientes.

> La función de la biblioteca estándar *uncaught_exception()*, declarada en `<exception>`, devuelve `true` si una excepción se ha lanzado pero no se ha capturado aún. Esto permite al programador especificar acciones diferentes en un destructor dependiendo de si un objeto se está destruyendo normalmente o como parte del proceso del mecanismo de excepciones.

5.3.4. La función *unexpected()*

Si una especificación de excepción, o lista `throw` (§5.2.3) *miente*, el compilador castigará nuestro pecado llamando automáticamente a la función de la biblioteca estándar *unexpected()*.

Esta función está implementada como un puntero a función, de forma que podemos cambiar su comportamiento predefinido, que consiste en llamar a *terminate()* (vea §5.3.3); para ello haremos una llamada a la función *set_unexpected()*, declarada en `<exception>`, pasándole la dirección de una función que no reciba ni devuelva nada. Asimismo, *set_unexpected()* devuelve la dirección de la función anterior que estamos reemplazando.

Nuestra función *unexpected()* no debe recibir ni devolver nada, y además no debe regresar: debe acabar llamando a una función que haga terminar el programa, como *exit()* o *abort()*, pero también, y en esto se diferencia de *terminate()*, puede lanzar una excepción, incluso relanzar la misma. En este caso, la búsqueda de su manejador empieza en la llamada a función que lanzó la excepción inesperada.

Ejemplo:

excep-unexpec.cpp

```
1  /*
2   * Especificaciones de excepciones y unexpected()
3   */
4  #include <exception>
5  #include <iostream>
6  #include <cstdlib>
7  #include <cstring>
8  using namespace std;
9
10 class Arriba {};
11 class Abajo {};
12 void g();
13
14 void f(int i) throw(Arriba, Abajo)
15 {
16   switch (i) {
17   case 1: throw Arriba();
18   case 2: throw Abajo();
19   }
20   g();
21 }
22
23 #ifndef UNEXPECTED
24 void g() {} // Versión 1: correcto
25 #else
26 void g() { throw 666; } // Versión 2: excepción inesperada en f()
27 #endif
28
29 void el_inesperado()
30 {
31   cerr << "Se ha lanzado una excepción a traición." << endl;
32   exit(EXIT_FAILURE);
33 }
34
35 int main()
36 {
37   (void) set_unexpected(el_inesperado);
38   for (int i = 1; i <= 3; i++) try {
39     f(i);
40   } catch(Arriba) {
41     cout << "Arriba capturado." << endl;
42   } catch(Abajo) {
43     cout << "Abajo capturado." << endl;
44   }
45 }
```

46

La función *f()* promete que sólo va a lanzar excepciones de los tipos *Arriba* y *Abajo*, y eso es lo que parece mirando su código. La versión 1 de *g()* no lanza nada, así que es verdad. Sin embargo, algún día alguien cambia *g()* de forma que ahora lanza una excepción de tipo *int*, y enlaza *g()* con *f()* (podrían estar en ficheros separados). Ahora *f()* lanzará también indirectamente una excepción de tipo *int*, violando su promesa, o sea, su lista `throw`, aunque quizá sin el conocimiento de su autor, y provocando una llamada a *unexpected()*.

Esto nos enseña que siempre que empleemos excepciones y listas `throw`, deberíamos escribir nuestra versión de *unexpected()*, como en este ejemplo, para al menos registrar el suceso y que nos demos cuenta, y además relanzar el mismo error, lanzar otro nuevo, o terminar el programa lo mejor posible, como hacemos aquí.

Otra forma de abordar el problema del lanzamiento de excepciones que no estén en la lista `throw` es añadir a ésta la excepción estándar *bad_exception* (§5.4.1), declarada en `<exception>`. En este caso, *unexpected()* lanzará esta excepción en lugar de llamar a una función.

Ejemplo:

```
class X { };
class Y { };
void f() throw(X, std::bad_exception)
{
  // ...
  throw Y(); // lanza una excepción «mala»
}
```

En este caso, la función *f()* no acaba lanzando la excepción *Y*, ni *unexpected()* llama a la función instalada con *set_unexpected()* o, si no se ha instalado ninguna, a *terminate()*, sino que se lanza la excepción estándar *bad_exception*, que será la que habrá que capturar. Obsérvese que de todas formas se pierde información incluso de qué excepción es la que realmente se ha producido.

5.4. Jerarquía de excepciones

Una excepción es un objeto de alguna clase (también puede ser de un tipo fundamental) que representa la ocurrencia de un suceso excepcional, que

normalmente se ve como un problema o error. A menudo, las excepciones se pueden agrupar de forma natural en familias. Esto implica que la herencia puede ser un mecanismo útil a veces para estructurarlas. Por ejemplo, las excepciones para una biblioteca matemática podrían estructurarse así:

```
class ErrorMatematico { };
class DesbordamientoSuperior: public ErrorMatematico { };
class DesbordamientoInferior: public ErrorMatematico { };
class DivisionPorCero: public ErrorMatematico { };
// ...
```

Esto nos permitiría manejar cualquier error matemático sin importarnos de qué clase sea. Por ejemplo:

```
void f()
{
  // ...
  try {
    // ...
  } catch(DesbordamientoSuperior) {
     // manejo de DesbordamientoSuperior o algo derivado de él
  } catch(ErrorMatematico) {
     // manejo de cualquier otro error matemático (no el anterior)
  }
}
```

Si estas excepciones se hubieran expresado de forma independiente unas de otras, sin agruparse en esa jerarquía, una función que quisiera poder capturar todas las excepciones de nuestra biblioteca matemática tendría que listar todos los manejadores, uno por excepción. Esto es tedioso y propenso a olvido de alguna excepción. Pero además, requeriría añadir otro manejador a la lista si por algún motivo se añadiera otra excepción a la biblioteca.

El empleo de jerarquías de excepciones lleva a manejadores interesados sólo en un subconjunto de la información expresada por las excepciones. O sea, que una excepción se captura por un manejador para su clase base en lugar de para su tipo exacto. En este caso, si no se emplean punteros ni referencias, parte de la información de la excepción lanzada se pierde en el proceso de la captura. Por ejemplo:

```
class ErrorMatematico {
  // ...
public:
  virtual void ver_traza() const { cerr << "Error matemático"; }
};
```

```
class Desbordamiento_int: public ErrorMatematico {
  const char* op;
  int a1, a2;
  // ...
public:
  Desbordamiento_int(const char* p, int a, int b)
  : op(p), a1(a), a2(b) { }
  virtual void ver_traza() const {
    cerr << op << '(' << a1 << ", " << a2 << ')';
  };

void f()
{
  try {
    g();
  } catch(ErrorMatematico m) {
      // ...
  }
}
```

Cuando se alcanza el manejador para *ErrorMatematico*, *m* es un objeto de este tipo, aunque la llamada a *g()* haya provocado un error del tipo derivado *Desbordamiento_int*, con lo cual la información extra de este tipo se pierde.

Para evitar esta pérdida de información, deben emplearse punteros o referencias. Por ejemplo:

```
int sumar(int x, int y)
{
  if ((x > 0 && y > 0 && x > INT_MAX - y)
      ||
      (x < 0 && y < 0 && x < INT_MIN - y))
    throw Desbordamiento_int("+", x, y);
  return x + y;
}

void f() try
{
  int i1 = sumar(1, 2);
  int i2 = sumar(INT_MAX, -2);
  int i3 = sumar(INT_MAX, 2);  // excepción
} catch(ErrorMatematico& m) {
    // ...
    m.ver_traza();
}
```

En este caso se llama en el manejador a *Desbordamiento_int::ver_traza()*; si se hubiera capturado por valor, se hubiera llamado en cambio a la función de la clase base: *ErrorMatematico::ver_traza()*.

5.4.1. Excepciones estándares

Algunos operadores del lenguaje C++, y el propio lenguaje, pueden lanzar excepciones. Algunos métodos de clases de la biblioteca estándar también. Estas excepciones y otras más que se definen aunque no las lance ni el lenguaje ni la biblioteca estándar están disponibles además para quien quiera emplearlas. Hay quien opina (Stroustrup no) que estas excepciones proporcionan una forma fácil y rápida de emplear excepciones sin tener que definir unas propias; también se pueden definir excepciones derivándolas de éstas.

Las excepciones estándar forman una jerarquía. A continuación se describen brevemente. El sangrado indica de quién deriva cada clase.

exception Definida en `<exception>`. Es la clase base para todas las excepciones estándares. Posee un método virtual llamado *what()* que redefinen las otras excepciones derivadas de ésta. Este método devuelve una cadena de caracteres de bajo nivel que describe la excepción o da su nombre.

- **logic_error** Definida en `<stdexcept>`. Se supone que los errores lógicos podrían detectarse antes de la ejecución del programa o comprobando los parámetros pasados a funciones y constructores.
  - **length_error** Definida en `<stdexcept>`. Indica un intento de producir un objeto cuya longitud sea mayor o igual que *npos* (el valor más grande representable en un *size_t*).
  - **domain_error** Definida en `<stdexcept>`. Para informar de violaciones de precondiciones.
  - **out_of_range** Definida en `<stdexcept>`. Lanzada por el método *at()* de varias clases contenedoras como *string* o *vector* y por el operador de índice *operator* `[]` de *bitset* para informar de un parámetro cuyo valor está fuera de un rango válido.
  - **invalid_argument** Definida en `<stdexcept>`. Lanzada por el constructor de *bitset*, indica un parámetro inválido para la función desde donde se lanza.
- **bad_alloc** Definida en `<new>`. Lanzada por el operador `new` en caso de fallo en la petición de memoria, si no se ha instalado una función para tratar el caso mediante la función *set_new_handler()* y si no se ha pasado el argumento *nothrow*, que haría que `new` devolviera 0.
- **bad_exception** Definida en `<exception>`. Lanzada por el lenguaje cuando una función lanza una excepción que no está en su lista `throw` pero ésta contiene precisamente *bad_exception* (p. 328).

ios_base::failure Definida en `<ios>`. Lanzada por *ios_base::clear()* para indicar un fallo de entrada/salida.

bad_typeid Definida en `<typeinfo>`. Lanzada por el operador de identificación de tipo en tiempo de ejecución (RTTI, *Run Time Type Identification*) `typeid` cuando se le pasa un puntero nulo.

bad_cast Definida en `<typeinfo>`. Lanzada por el operador de RTTI `dynamic_cast` cuando se le pasa una referencia que no es del tipo esperado.

runtime_error Definida en `<stdexcept>`. Para informar de errores que se producirán sólo cuando el programa esté ejecutándose.

- **range_error** Definida en `<stdexcept>`. Para informar de violaciones de una postcondición.
- **overflow_error** Definida en `<stdexcept>`. Lanzada por el método *bitset::to_ulong()*, para informar de un desbordamiento superior matemático.
- **underflow_error** Definida en `<stdexcept>`. Para informar de un desbordamiento inferior matemático.

Para más claridad se presentan las excepciones estándar lanzadas por el lenguaje en la tabla 5.1 y las lanzadas por la biblioteca estándar en la tabla 5.2.

Excepción	Lanzada por	Cabecera
bad_alloc	`new`	`<new>`
bad_cast	`dynamic_cast`	`<typeinfo>`
bad_typeid	`typeid`	`<typeinfo>`
bad_exception	*lista* `throw`	`<exception>`

Cuadro 5.1: Excepciones estándares lanzadas por el lenguaje C++

Excepción	Lanzada por	Cabecera
out_of_range	*at()* *bitset::operator[]()*	`<stdexcept>`
invalid_argument	*constructor de bitset*	`<stdexcept>`
overflow_error	*bitset::to_ulong()*	`<stdexcept>`
ios_base::failure	*ios_base::clear()*	`<ios>`

Cuadro 5.2: Excepciones estándares lanzadas por la biblioteca estándar

La jerarquía de excepciones estándares se verá mejor en la figura 5.1.

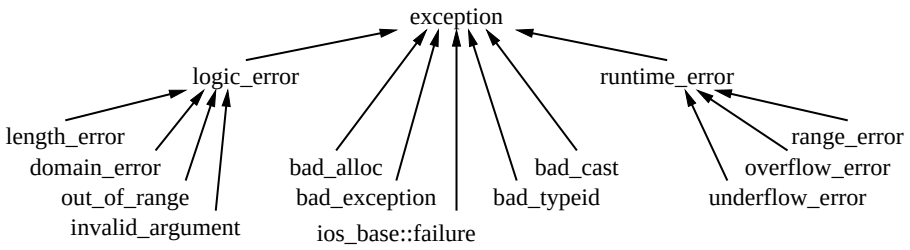

Figura 5.1: Jerarquía de excepciones estándares

5.5. Programación con excepciones

Las excepciones no estaban en el C++ original; se añadieron un tiempo después. A continuación se presentan unos consejos y advertencias: cuándo, cuándo no, y cómo emplear excepciones.

5.5.1. Evitar excepciones

Las excepciones no son la panacea o la solución a todos los problemas. A veces es mejor no emplearlas. He aquí cuándo.

No para eventos asíncronos

El sistema de señales basado en la función de la biblioteca estándar de C *signal()* maneja eventos o sucesos asíncronos; es decir, aquéllos que ocurren fuera del ámbito del programa y que por tanto éste no puede preveer. Las excepciones no sirven para este tipo de sucesos. Sin embargo pueden *asociarse* a ellas. El manejador de señal puede hacer su trabajo rápidamente, regresar, y luego el programa puede lanzar una excepción basándose en alguna variable puesta por este manejador de señal.

No para errores ordinarios

Si se posee suficiente información como para tratar un error, no es una excepción. Si el error se puede tratar en el contexto en el que se produce, hágase ahí.

Tampoco son excepciones de C++ los sucesos a nivel de máquina como división por cero; éstos se manejan mediante otros mecanismos, proporcionados por el sistema operativo o la circuitería (aunque también pueden asociarse a excepciones).

No para control de flujo

Las excepciones pueden verse como otro mecanismo de control de flujo, una especie de mezcla entre `return` y `switch`. Por lo tanto podrían emplearse sin que se produzcan realmente errores o problemas en el programa: excepciones que no son errores.

No obstante, esto suele ser una mala idea. En parte porque las excepciones son mucho menos eficientes que la ejecución normal del programa: las excepciones son sucesos raros, excepcionales, y el programa, en ausencia de ellos, no debería verse afectado en su rendimiento. Y en parte porque emplear excepciones para lo que no han sido diseñadas es confuso.

No si se pueden evitar

No es obligatorio emplear excepciones. Algunos programas, como se dijo en 5.1, son bastante (relativamente) simples: se toma una entrada, se procesa de alguna forma, y se muestra una salida. Es posible que falte memoria, que no se pueda abrir algún fichero o que el usuario haya dado algún parámetro incorrecto. Es perfectamente válido y recomendable en este tipo de programas presentar un mensaje de diagnóstico y acabar ordenadamente; incluso emplear *assert* aunque sea en la fase de pruebas, o *abort()*. No valdría la pena trabajar extra para meter excepciones cuando lo más fácil para el usuario es volver a ejecutar el programa.

No para código antiguo

Si se tiene que modificar un programa antiguo que no emplea excepciones y se le añade una biblioteca que sí las utiliza, ¿habrá que modificar todo el programa para adaptarlo a las excepciones? La respuesta es no, o no mucho. Quizás se pueda aislar el código que genere excepciones en un bloque `try` con manejadores que conviertan las excepciones al esquema de manejo de errores que tuviera el programa antiguo. En cualquier caso se podría rodear de un bloque `try` la porción de programa más grande que pudiera generar excepciones (esto puede ser *main()*) seguido de un manejador para cualquier excepción (`catch(...)`) que generara mensajes básicos de error. Esto podría refinarse más adelante según se viera, pero el código a añadir es muy poco.

5.5.2. Empleo de excepciones

A continuación se expone qué cosas se pueden hacer con excepciones, o para qué sirven.

- Corregir el problema y llamar de nuevo a la función que lo causó.
- Corregir el problema y seguir, sin volver a llamar a la función que lo causó.
- Calcular algún resultado alternativo en vez del que la función se supone que produciría.
- Hacer lo que se pueda en el contexto en curso y relanzar la misma excepción a uno superior.
- Hacer lo que se pueda en el contexto en curso y relanzar otra excepción diferente a uno superior.
- Terminar el programa ordenadamente.
- Envolver funciones (por ejemplo y especialmente de alguna biblioteca de C) que empleen otros esquemas de error, de forma que generen excepciones.
- Simplificar. El empleo de excepciones debe ser más fácil, claro y cómodo que otro que se pueda usar.
- Hacer una biblioteca o programa más seguro y robusto.

Y por último algunos consejos:

Emplee siempre especificaciones de excepción Las especificaciones de excepción o listas `throw` son como prototipos de funciones: les dicen al usuario que le conviene escribir manejadores de excepción, y cuáles, y les dicen al compilador qué excepciones pueden lanzarse desde la función.

Empiece con las excepciones estándar Antes de crear sus excepciones, estaría bien que mirara a ver si alguna de las que ya hay (§5.4.1) le sirve. Si es así, es posible que sea más fácil para el usuario reconocerla y tratarla. Si no, puede intentar derivar la suya de una de las existentes o directamente de *exception*. Así el usuario siempre podrá esperar llamar en sus manejadores al método *what()* definido en la clase de interfaz *exception*.

Anide sus propias excepciones Si se crea excepciones para una clase particular, anídelas en ella (esto es, defínalas dentro); esto le dice claramente al usuario que esas excepciones sólo son para esa clase; además impide la proliferación de nombres en el espacio de nombres global o en el que se defina la clase anfitriona. Este anidamiento puede hacerse aunque las excepciones deriven de las estándares.

Emplee jerarquías de excepciones Pues proporcionan un modo muy conveniente de clasificar los diversos tipos de errores que puedan encontrarse en la clase o biblioteca que se esté construyendo. Dan información valiosa a los usuarios, les asisten en la organización de su código, y les da la opción de capturar solamente el tipo base, despreciando todos los tipos específicos de error; además, al añadir una nueva excepción a la jerarquía, ésta se capturará en el manejador de la clase base, sin necesidad de tocar el código cliente con los manejadores.

Precisamente las excepciones estándar son un buen ejemplo de esto (fig. 5.1).

Herencia múltiple Uno de los pocos sitios donde la herencia múltiple puede estar justificada es precisamente en las jerarquías de excepciones, porque un manejador de una clase base de excepción de cualquiera de las raíces de la clase de excepción heredada múltiplemente puede manejar dicha excepción.

Ejemplo:

```
class E_Fichero_NFS: public E_Red, public E_SistemaFicheros {
  // ...
};
```

Un error relativo a un fichero remoto (*E_Fichero_NFS*) puede ser capturado en funciones que traten con excepciones o errores de red (*E_Red*):

```
void f()
try {
  // ...
} catch(E_Red& e) {
    // ...
}
```

tanto como por funciones que traten con excepciones o errores del sistema de ficheros (*E_SistemaFicheros*):

```
void g()
try {
  // ...
} catch(E_SistemaFicheros& e) {
    // ...
}
```

Esto es importante en casos donde haya servicios transparentes al usuario; obsérvese que el autor de *g()* puede no saber o no importarle que hay una

red por medio; sin embargo, su manejador también capturará los errores de red (*E_Fichero_NFS*).

Capture por referencia, no por valor Si se lanza un objeto de una clase derivada y se captura por valor en el manejador de la clase base, los miembros añadidos en la derivación se pierden[2] y el objeto copiado en el parámetro formal del manejador se comportará como un objeto de la clase base.

En vez de por referencia, se podrían lanzar y capturar punteros, pero esto añadiría mayor complejidad. El lanzador y el capturador tendrían que ponerse de acuerdo en cómo el objeto de excepción obtiene y libera memoria.

De todas formas, aunque no haya una jerarquía de excepciones, siempre suele ser mejor capturar por referencia, debido al ahorro de una copia.

Lance excepciones en constructores Éstos son uno de los sitios más importantes donde lanzar excepciones, pues son la mejor alternativa al tratamiento de errores, ya que la construcción (correcta) de objetos es fundamental en C++. El lanzamiento de una excepción en un constructor garantiza que el objeto no está construido (lo cual es diferente de que esté mal construido). Sin embargo, hay que tener cuidado cuando el objeto contiene punteros para los que se reserve memoria en el constructor.

No lance o cause excepciones en destructores Puesto que se llama a destructores en el proceso de lanzar otras excepciones, si se lanza una excepción en un destructor, se lanzará *antes* de que se alcance la cláusula `catch` de la excepción originalmente lanzada, lo que provocará una llamada a *terminate()*.

Y si en el destructor se llamara a alguna función que pudiera lanzar excepciones, debería meterse su llamada en un bloque `try` de forma que ninguna excepción escapara del destructor.

Evite punteros «desnudos» Un simple puntero significa vulnerabilidad en el constructor si se reserva memoria u otro recurso para él. Puesto que un puntero no posee destructor, esos recursos no serán liberados en el caso de que se lance una excepción en el constructor, como se ha mencionado antes.

> A este respecto, la biblioteca estándar viene en nuestra ayuda proporcionándonos un tipo pseudo-puntero llamado *auto_ptr*, definido en

[2] A esto se le llama *rebanar* el objeto.

`<memory>`, que posee la curiosa e interesante propiedad de que el objeto al que apunte será implícita y silenciosamente borrado cuando el objeto *auto_ptr* salga de ámbito.

Ejercicios

E5.1. Rehaga el ejercicio **E4.6** de la clase *Fecha*, pero asegurándose siempre de que cualquier objeto de esa clase contenga una fecha correcta. Emplee, por supuesto, excepciones. Escriba un programa de prueba donde pruebe y capture las excepciones. Por ejemplo, pida una fecha desde la entrada estándar y asegúrese de que la fecha introducida sea correcta, de forma que si no lo es se presente un mensaje y se vuelva a pedir hasta que lo sea. Repita de forma que si la entrada es incorrecta, la fecha almacenada sea la de *hoy*.

PISTA: Cree una clase *FechaInvalida* dentro de *Fecha*.

Cree en el programa de prueba dos funciones que devuelvan una *Fecha* correcta leída desde la entrada estándar.

E5.2. Modifique el ejercicio **E4.9** de forma que el *operator* `[]` lance una excepción cuando el índice que se le pase esté fuera del rango correcto. Cambie el programa de prueba para que haga uso de esa excepción. Considere si crear una clase de excepción nueva o derivada de alguna estándar existente, o si simplemente podría emplear alguna de éstas.

E5.3. Escriba un programa para probar que si crea un objeto de excepción en memoria dinámica y lanza un puntero a ese objeto, dicho objeto *no* será destruido al acabar su captura. ¿Qué tendría que hacer entonces para destruirlo?

E5.4. Siga la pista a la creación y paso de una excepción empleando una clase para ella, con un constructor y un constructor de copia que se anuncien a sí mismos y proporcionen tanta información como sea posible sobre el objeto que se esté creando (y en el caso del constructor de copia, sobre el objeto del cual se está creando la copia). Escriba un programa de prueba donde se lance un objeto del tipo creado, y analice el resultado.

Capítulo 6

La Biblioteca Estándar de E/S (IOStreams)

6.1. La Biblioteca Estándar de C++

Un conjunto de componentes (clases, funciones, variables externas, etc) reutilizables y que se organizan en módulos para formar un *archivo*, se denomina *biblioteca*.

El editor de enlaces es el que se encarga de extraer los componentes necesarios de una biblioteca para incorporarlos al fichero que contendrá la imagen ejecutable (bibliotecas estáticas) o para cargarlos en memoria cuando el programa ya se está ejecutando (bibliotecas dinámicas).

El lenguaje C++, al igual que C, no es capaz por sí mismo de realizar E/S, ni de acceder a la fecha del sistema, etc. Todo esto se lleva a cabo mediante clases y funciones definidas empleando los mecanismos básicos del lenguaje y, quizás, otros dependientes del sistema.

Poco transportable sería un programa, y de nada serviría tener un lenguaje normalizado, si para hacer cualquier cosa hubiera que utilizar una clase o una función proporcionada por un fabricante, y cada uno de ellos tuviera una distinta para hacer lo mismo.

Por tanto resultó evidente para el comité encargado de la normalización del lenguaje que también había que encargarse de un conjunto de componentes mínimo pero lo más general y útil posible que conformara la biblioteca estándar de C++.

Esta biblioteca principalmente proporciona:

- Apoyo al lenguaje en la gestión de la memoria y la identificación de

tipos en tiempo de ejecución.

- Información sobre aspectos del lenguaje dependientes del sistema[1].
- Funciones que no se pueden implementar de manera óptima en el propio lenguaje para todos los sistemas[2].
- Componentes no predefinidos en cuya transportabilidad se puede confiar (flujos de E/S, estructuras de datos, algoritmos, etc.).
- Un marco para facilitar su propia extensión.

Los componentes de la biblioteca estándar se definen en el espacio de nombres *std* y se declaran a través de un conjunto de cabeceras. Las cabeceras identifican las partes principales de la biblioteca. Su enumeración, por tanto, proporciona una visión de conjunto de la biblioteca. Por ello, a continuación se va a dar una lista de las cabeceras agrupadas por funcionalidad:

- Contenedores, tabla 6.1.
- Utilidades generales, tabla 6.2.
- Iteradores, tabla 6.3
- Algoritmos, tabla 6.4.
- Diagnóstico, tabla 6.5.
- Cadenas, tabla 6.6.
- Entrada/Salida, tabla 6.7.
- Localización, tabla 6.8.
- Utilidades de apoyo, tabla 6.9.
- Números, tabla 6.10.

Una cabecera estándar cuyo nombre comienza por la letra *c* es equivalente a una cabecera de la biblioteca estándar de C. Para cada cabecera `<cX>` que declara un nombre en el espacio de nombres *std* hay una cabecera `<X.h>` que declara el mismo nombre en el espacio de nombres global.

Los contenedores asociativos *multimap* y *multiset* se pueden encontrar en `<map>` y `<set>` (tabla 6.1), respectivamente; *priority_queue* se define en `<queue>`.

[1] Como el rango de los números enteros.
[2] Por ejemplo, *sqrt()* y *memmove()*.

Cabecera	Descripción
`<vector>`	vectores
`<list>`	listas doblemente enlazadas
`<dequeue>`	colas de doble extremo
`<queue>`	colas
`<stack>`	pilas
`<set>`	conjuntos
`<map>`	aplicaciones
`<bitset>`	secuencias de bits

Cuadro 6.1: Contenedores

Cabecera	Descripción
`<utility>`	operadores y pares
`<functional>`	objetos función
`<memory>`	asignadores para contenedores
`<ctime>`	fecha y hora

Cuadro 6.2: Utilidades generales

La cabecera `<memory>` (tabla 6.2) contiene también la plantilla *auto_ptr* que se utiliza fundamentalmente para facilitar la interacción entre punteros y excepciones.

Cabecera	Descripción
`<iterator>`	iteradores

Cuadro 6.3: Iteradores

Un algoritmo genérico se puede aplicar a (casi) cualquier secuencia de cualquier tipo. Las funciones de la biblioteca estándar de C *bsearch()* y *qsort()* se aplican sólo a los vectores de bajo nivel con elementos sencillos (sin constructores de copia ni destructores definidos por el usuario).

La cabecera `<cstring>` (tabla 6.6) declara la familia de funciones *strlen()*, *strcpy()*, etc. `<cstdlib>` declara *atof()* y *atoi()* que convierten cadenas de bajo nivel en valores numéricos.

Los manipuladores (tabla 6.7) son objetos que se utilizan para manipular el estado de un flujo (por ejemplo, para cambiar el formato de salida en coma flotante).

locale() (tabla 6.8) localiza diferencias culturales como el formato de salidas para fechas, el símbolo utilizado para representar la moneda, etc.

Cabecera	Descripción
`<algorithm>`	algoritmos genéricos
`<cstdlib>`	*bsearch()* y *qsort()*

Cuadro 6.4: Algoritmos

Cabecera	Descripción
`<stdexcept>`	excepciones estándar
`<cassert>`	macro *assert*
`<cerrno>`	manejo de errores, *errno*

Cuadro 6.5: Diagnóstico

La cabecera `<cstddef>` (tabla 6.9) define el tipo de valores devueltos por `sizeof`, *size_t*, el tipo del resultado de la sustracción de punteros, *ptrdiff_t*, y la macro *NULL*, que en C++ no se debe utilizar.

Por razones históricas, *abs()*, *fabs()* y *div()* se encuentran en `<cstdlib>` (tabla 6.10) y no en `<math>` con el resto de las funciones matemáticas.

Para utilizar un componente de la biblioteca estándar hay que incluir su cabecera. Escribir directamente las declaraciones pertinentes no es una alternativa que se ajuste al estándar. La razón es que algunas implementaciones optimizan el proceso de compilación cuando se incluyen cabeceras estándar.

6.2. La Biblioteca Estándar de E/S

Hasta ahora se ha utilizado solamente E/S elemental: lectura de tipos básicos desde la entrada estándar y su escritura por la salida estándar, mediante los operadores de inserción y extracción. Ahora se tratará todo esto en más detalle.

Cabecera	Descripción
`<string>`	cadenas (*string*)
`<cctype>`	clasificación de caracteres
`<cwtype>`	clasificación de caracteres anchos
`<cstring>`	funciones de cadenas de bajo nivel
`<cwchar>`	funciones de cadena de caracteres anchos
`<cstdlib>`	funciones de cadenas de bajo nivel

Cuadro 6.6: Cadenas

Cabecera	Descripción
`<iosfwd>`	declaraciones adelantadas de componentes de E/S
`<iostream>`	bases de *iostream*
`<ios>`	objetos y operaciones de flujos
`<streambuf>`	búferes de flujo
`<istream>`	flujos de entrada
`<ostream>`	flujos de salida
`<iomanip>`	manipuladores
`<sstream>`	flujos de cadenas (*string*)
`<cctype>`	funciones de tipos de caracteres
`<fstream>`	flujos de fichero
`<cstdio>`	E/S estilo C
`<cwchar>`	E/S de caracteres anchos

Cuadro 6.7: Entrada/Salida

Cabecera	Descripción
`<locale>`	localización
`<clocale>`	localización estilo C

Cuadro 6.8: Localización

En C++, como en C, no hay nada incorporado en el lenguaje sobre E/S. Todas las funciones de E/S deben ser definidas. Como la E/S es muy dependiente del sistema, se espera que en cada uno el fabricante suministre las funciones más o menos básicas que permitan trabajar con ficheros, consola, terminal, etc. Estas funciones suelen agruparse para formar bibliotecas.

Afortunadamente la biblioteca de E/S de Unix, por ser la más sencilla, fue adaptada por el comité ANSI de C y adoptada como estándar. En ella se definía el concepto de *flujo* de datos: una corriente de *bytes* que actúa como fuente o destino de datos según sea de entrada o de salida.

Todas las operaciones corrientes se pueden hacer mediante los flujos; por muy complicado que sea el soporte que proporcione el sistema operativo para los distintos periféricos, es la biblioteca la que se encarga de las transformaciones. El flujo actúa como una interfaz entre el programador y el sistema operativo. Cuando se desee más control o rapidez, habrá que utilizar las funciones proporcionadas por el sistema.

En C++, heredero de C, puede seguirse empleando la biblioteca de C de E/S: no hay más que incluir la ya conocida cabecera `<cstdio>` y utilizar las conocidas funciones. Sin embargo, pocos son los que, proponiéndose programar

Cabecera	Descripción
`<limits>`	límites numéricos
`<climits>`	macros de límites numéricos enteros
`<cfloat>`	macros de límites numéricos de coma flotante
`<new>`	gestión de la memoria dinámica
`<typeinfo>`	identificación de tipos en tiempo de ejecución
`<exception>`	manejo de excepciones
`<cstddef>`	operadores y tipos
`<cstdarg>`	funciones con Nº variable de parámetros
`<csetjmp>`	manipulación de pila
`<cstdlib>`	terminación del programa
`<ctime>`	manejo del tiempo
`<csignal>`	manejo de señales

Cuadro 6.9: Utilidades de apoyo

Cabecera	Descripción
`<complex>`	números complejos y sus operaciones
`<valarray>`	vectores numéricos y sus operaciones
`<numeric>`	algoritmos numéricos genéricos
`<cmath>`	funciones matemáticas estándar
`<cstdlib>`	números aleatorios

Cuadro 6.10: Números

en serio en C++, utilizan esta biblioteca.

La biblioteca de E/S para C++ sigue conservando el concepto de flujo, como en C. ¿Cuáles son las ventajas de esta biblioteca sobre la de C?

- Mejor comprobación de los tipos de los datos de E/S. En las famosas funciones *printf()* y *scanf()* no hay comprobación de tipos, como debe saber: sólo el primer parámetro es conocido; el tipo y número de los siguientes, si los hay, se basan en la cadena de formato[3].
- Tratamiento uniforme de todos los tipos de datos. Esto incluye, y es muy importante, los definidos por el usuario.
- Capacidad de extensión. Se puede crear un nuevo flujo de datos, se pueden sobrecargar los operadores de E/S, etc.

[3]Aunque algunos compiladores analizan la cadena de formato y así comprobar el resto de los parámetros.

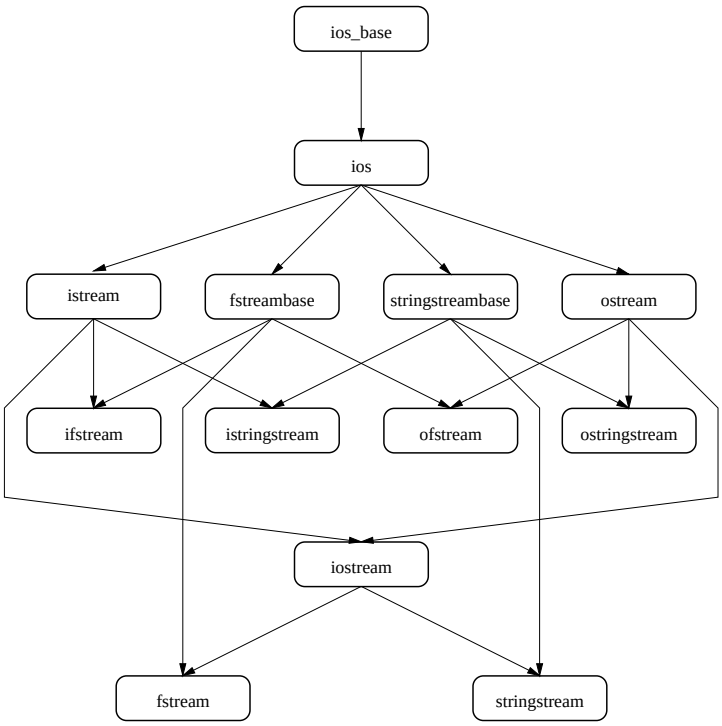

Figura 6.1: Jerarquía de clases de flujos

Para utilizar la biblioteca de E/S de C++ hay que incluir, fundamentalmente, el fichero de cabecera `<iostream>`, como se ha hecho hasta ahora. Se verá que a veces habrá que incluir también alguna otra cabecera, dependiendo de lo que se vaya a hacer.

No se pretende dar una descripción muy detallada de la biblioteca; para ello consúltese la bibliografía. Se expondrá aquí un resumen de sus capacidades, recalcando los aspectos más útiles.

En la figura 6.1 se muestra un esquema con la jerarquía de clases de los flujos; algunas de estas clases se verá a continuación.

6.3. Flujos de salida

En `<iostream>` se define la clase *ostream*, una especialización para una modalidad concreta de carácter de la plantilla *basic_ostream*.

```
typedef basic_ostream<char> ostream;
```

Esta clase plantilla *basic_ostream*, derivada de la clase *basic_ios* presente en `<ios>`, y sus operaciones asociadas de salida se definen en el espacio de nombres *std* y las presenta `<ostream>`, que contiene los componentes de `<iostream>` relacionados con la salida.

Algunos miembros públicos de esta clase son:

```
ostream& operator <<(bool);            // lógico
ostream& operator <<(int);             // entero
ostream& operator <<(double);          // real en doble precisión
ostream& operator <<(const void*);     // puntero, dirección
ostream& put(char c);                  // carácter
ostream& flush();                      // vacía el búfer
ostream& write(const char* s, streamsize n);
                                       // cadena de bajo nivel
                                       // (hasta n caracteres)
```

streamsize es un tipo entero con signo que se emplea para representar el número de caracteres transferidos en una operación de E/S y el tamaño de los búferes de E/S.

La salida se envía a un objeto de la clase *ostream*, lo que permite por ejemplo concatenar operaciones, como se ha hecho ya con el operador `<<`. Existen algunos objetos predefinidos que ya se conocen:

cout	Salida estándar normal
cerr	Salida estándar de errores
clog	Salida estándar de errores, pero el búfer se vacía sólo cuando está lleno

Ejemplo:

```
cout.put('A');                 // imprime A
const char* str = "ABCDEFGHI";
cout.write(str + 2, 3);        // imprime CDE
cout.flush();                  // vacía el búfer de salida
```

Hay que señalar que un valor *bool* se escribirá por omisión en la salida como 0 o 1. Si se desea obtener `true` o `false` se puede fijar el indicador de formato *boolalpha*[4].

[4]Esto en algunos compiladores todavía no está incluido

Ejemplo:

boolean.cpp

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

int main()
{
  cout << true << ' ' << false << '\n'
       << boolalpha       // para usar representación simbólica
       << true << ' ' << false << endl;
}
```

Esto imprimiría:

```
1 0
true false
```

Más precisamente, *boolalpha* garantiza que se obtendrá una representación de los valores *bool* dependiente de la *localización*.

6.3.1. El operador de inserción <<

El operador de desplazamiento de *bits* hacia la izquierda, `<<`, como ya se sabe, está sobrecargado para trabajar con los tipos básicos. Su asociatividad y precedencia es la misma que la del operador incorporado en el lenguaje, lo cual es conveniente la mayoría de las veces. Por ejemplo,

```
cout << "x = " << x << "\n";
```

equivale a

```
((cout.operator<<("x = ")).operator<<(x)).operator("\n");
```

Además, la precedencia de `<<` es lo suficientemente baja como para permitir expresiones aritméticas como operandos sin utilizar paréntesis.

Ejemplo:

```
cout << "a * b + c = " << a * b + c << '\n';
```

Aunque, a veces, la precedencia da algún problema:

```
cout << a + b << " es una suma\n";               // bien
cout << a & b << " es un problema\n";            // ERROR
cout << (a & b) << " es un problema resuelto\n"; // bien
```

Se puede utilizar el operador de desplazamiento a la izquierda en una instrucción de salida pero, naturalmente, debe aparecer entre paréntesis:

```
cout << "a << b = " << (a << b) << '\n';
```

6.3.2. Salida formateada

Campos de salida

A menudo se desea rellenar con texto un espacio específico de una línea de salida, o se quieren utilizar n caracteres y no menos (y más solamente si no encaja el texto). Para hacerlo, se especifican una anchura de campo y un carácter de relleno:

```
class ios_base {
public:
  // ...

  streamsize width() const;  // obtener anchura de campo
  streamsize width(streamsize ancho);
                             // establecer anchura de campo
  // ...
};

class ios: public ios_base {
public:
  // ...

  char fill() const;         // obtener carácter de relleno
  char fill(char c);         // establecer carácter de relleno

  // ...
};
```

La función *width()* especifica el número mínimo de caracteres a utilizar para la siguiente operación de salida `<<` (*bool*, cadena de bajo nivel, carácter, puntero, *string*).

Ejemplo:

campo-salida.cpp

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

int main()
{
  cout.width(4);
  cout << 10;
}
```

Esto imprimiría 10 precedido de dos espacios: ␣␣10.

El carácter de relleno se puede especificar con la función *fill()*.

Ejemplo:

relleno-salida.cpp

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

int main()
{
  cout.width(4);
  cout.fill('#');
  cout << 10;
}
```

Esto imprimirá: ##10.

El carácter de relleno por omisión es el de espacio y la anchura por omisión es 0, que significa: tantos caracteres como sean necesarios.

El tamaño de campo se puede reestablecer a su valor por omisión de la siguiente forma:

```
cout.width(0);
```

La función *width()* establece el número mínimo de caracteres. Si se proporcionan más caracteres se imprimirán todos.

Ejemplo:

```
cout.width(4);
cout << 12345;
```

imprimiría 12345 y no sólo 1234.

Una llamada a `width(n)` afecta sólo a la operación de salida `<<` inmediatamente posterior.

Ejemplo:

```
cout.width(4);
cout.fill('#');
cout << 12 << ':' << 13;
```

Esto produce `##12:13`, en lugar de `##12:##13`.

Los manipuladores que se verán a continuación proporcionan una forma más elegante de especificar la anchura de un campo de salida.

Ajuste de campo

El ajuste de los caracteres dentro de un campo se puede controlar mediante llamadas a *setf()*:

```
cout.setf(ios_base::left, ios_base::adjustfield);     // izquierda
cout.setf(ios_base::right, ios_base::adjustfield);    // derecha
cout.setf(ios_base::internal, ios_base::adjustfield); // interno
```

Esto establece el ajuste de la salida en un campo de salida definido por `ios_base::width()` sin efectos colaterales sobre otras partes del estado del flujo.

Ejemplo:

ajuste-salida.cpp

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
```

```
{
  cout.fill('#');
  cout << '(';
  cout.width(4);
  cout << -10 << "), (";

  cout.width(4);
  cout.setf(ios_base::left, ios_base::adjustfield);
  cout << -10 << "), (";

  cout.width(4);
  cout.setf(ios_base::internal, ios_base::adjustfield);
  cout << -10 << ')' << endl;
}
```

Esto imprimiría: `(#-10), (-10#), (-#10)`

Manipuladores

El operador `<<` produce el mínimo número de caracteres; por ejemplo:

```
cout << 2.0;                          // imprime 2 y no 2.0

int i = 6, j = 7;
cout << i << j;                       // imprime 67
cout << i << " " << j;                // mejor: imprime 6 7
```

El método consistente en imprimir espacios, tabuladores y nuevas líneas donde haga falta puede que sea suficiente en algunos casos concretos, pero a veces se echan de menos algunas capacidades de formato como las que tenía *printf()*.

No obstante, en la biblioteca de E/S en C++ se dispone de los *manipuladores*, que son valores o funciones que, sin imprimir nada visible, producen algún efecto especial en el flujo en el que operan. Ya se ha visto uno, *endl*, que escribía `'\n'` y vaciaba el búfer de salida. De esta forma, ya no hay que manejar el estado de un flujo en términos de indicadores, sino a través de los manipuladores.

Los manipuladores más normales se recogen en la tabla 6.11. Otros (justo los que actúan como funciones, y reciben parámetros por tanto) requieren la inclusión de otra cabecera estándar: `<iomanip>`. Éstos se recogen en la tabla 6.12.

Los *bits* de formato de los manipuladores *(re)setiosflags()* se pueden nombrar con las constantes enumeradas de la clase *ios* que se muestran en la tabla 6.13

Manipulador	Descripción
`endl`	Nueva línea, vacía el búfer de salida
`ends`	Carácter nulo (`'\0'`, terminador de cadena)
`flush`	Vacía el búfer de salida
`dec`	Números en base decimal
`oct`	Números en base octal
`hex`	Números en base hexadecimal
`ws`	Saltar espacios en blanco en la entrada
`uppercase`	'E' de exponente y letras hexadecimales en mayúsculas
`nouppercase`	'e' de exponente y letras hexadecimales en minúsculas

Cuadro 6.11: Manipuladores en `<ios>`, `<ostream>` e `<iostream>`

Manipulador	Descripción
`setw(int)`	Anchura de campo
`setfill(char)`	Carácter de relleno
`setbase(int)`	Base de numeración
`setprecision(int)`	Precisión de reales
`setiosflags(ios_base::fmtflags)`	Establecer indicadores
`resetiosflags(ios_base::fmtflags)`	Borrar indicadores

Cuadro 6.12: Manipuladores en `<iomanip>`

Nombre	Estado	Significado
`skipws`	Sí	Saltar blancos en la entrada
`left`	No	Salida justificada a la izquierda
`right`	No	Salida justificada a la derecha
`internal`	No	Signo justificado a la izda., núm. a la derecha
`dec`	Sí	Base decimal. Equivale al manipulador `dec`
`oct`	No	Base octal. Equivale al manipulador `oct`
`hex`	No	Base hexadecimal. Equivale al manipulador `hex`
`showbase`	No	Emplear indicador de base: `0x` para `hex` y `0` para `oct`
`showpoint`	No	Poner punto o coma decimal y ceros decimales en números reales aunque sea superfluo
`uppercase`	No	Emplear mayúsculas para las letras de los números hexadecimales y el indicador de exponente de 10
`showpos`	No	Mostrar siempre el signo, aunque sea positivo
`scientific`	No	Forzar notación científica (exponencial) en números reales
`fixed`	No	Forzar notación fija en números reales
`unitbuf`	No	Vaciar todos los búferes tras la salida
`stdio`	No	Vaciar los búferes de *cout* y *cerr* tras la salida

Cuadro 6.13: Nombres de los *bits* de formato

(aparecen sus nombres, estado de activación predeterminado y significado). Para poner varias de ellas se empleará el operador de *bits* OR (`bit_or` o `|`); por ejemplo:

```
cout << setiosflags(ios_base::left |
                    ios_base::hex  |
                    ios_base::uppercase)
     << ...
```

Ejemplo:

El siguiente programa manipula enteros en diferentes bases (decimal, octal y hexadecimal). Observe que todos estos manipuladores también valen para la entrada.

bases.cpp

```
1  // Utilizando varias bases en la E/S de enteros
2
3  #include <iostream>   // <iomanip> no hace falta
4  using namespace std;
5
6  int main()
7  {
8    int i = 10, j = 16, k = 24;
9
10   cout << i << '\t' << j << '\t' << k << endl
11        << oct << i << '\t' << j << '\t' << k << endl
12        << hex << i << '\t' << j << '\t' << k << endl
13        << "Introduzca tres enteros;"
14        << " por ejemplo: 11 11 12a" << endl;
15   cin >> i >> hex >> j >> k;
16   cout << dec << i << '\t' << j << '\t' << k << endl;
17 }
```

La salida resultante para la entrada de ejemplo es:

```
10      16      24
12      20      30
a       10      18
Introduzca tres enteros; por ejemplo: 11 11 12a
11      17      298
```

6.3.3. Salida de tipos definidos por el usuario

El operador << se puede sobrecargar para insertar en un flujo cualquier tipo nuevo que se defina.

En vez de definir funciones miembro para mostrar los objetos de una clase, está más de acuerdo con la filosofía de C++ y de esta biblioteca el sobrecargar el operador << para ello: los tipos definidios por el usuario deberían saber mostrarse como los demás. Para ello se empleará el siguiente esqueleto:

```
ostream& operator <<(ostream& os, const T& obj)
{
  // lógica especial para el objeto de tipo T
  return os; // devolución del flujo de salida
}
```

El primer parámetro es una referencia a un flujo de salida; debido a esto, este operador debe ser definido como una función no miembro, fuera de la clase. Si debe tener acceso a sus miembros privados, como es común, debe ser declarado en ella como amigo. Devuelve el mismo flujo de salida como referencia para poder concatenar las operaciones.

Ejemplo:

Dada una clase para manejar números racionales véase cómo se puede sobrecargar el operador de inserción.

```
class Racional {
public:
  friend ostream& operator <<(ostream&, const Racional&);
  // ...
private:
  int n, d;
};

ostream& operator <<(ostream& salida, const Racional& r)
{
  salida << r.n;
  if (r.n != 0 && r.d != 1)
    salida << '/' << r.d;
  return salida;
}
```

6.4. Flujos de entrada

La entrada se maneja de forma similar a la salida. Como era de esperar el operador de desplazamiento de *bits* a la derecha se sobrecarga en la clase *istream* para los tipos básicos, y se llama entonces *extracción*. También se define un objeto de esta clase que representa al flujo de la entrada estándar: *cin*.

Análogamente a *basic_ostream*, *basic_istream* se define en `<istream>`, que contiene los componentes de `<iostream>` relacionados con la entrada.

Así, la clase *istream* se define como una especialización para una modalidad concreta de carácter de la plantilla *basic_istream*.

```
typedef basic_istream<char> istream;
```

Algunos miembros públicos de la clase *istream* son:

```
istream& operator >>(int&);    // entero
istream& operator >>(float&);  // real de simple precisión
istream& operator >>(double&); // real de doble precisión
int_type get();                // carácter, incluido blanco, y EOF
istream& get(char&);           // carácter, incluido blanco
istream& get(char*, streamsize, char d = '\n');
                               // cadena de caracteres
istream& getline(char*, streamzize, char d = '\n');
                               // cadena de caracteres
istream& read(char*, streamsize);
                               // cadena de caracteres
istream& putback(char);        // devuelve a la entrada el cter.
int peek();                    // toma el siguiente carácter
                               // sin extracción
int gcount();                  // da el número de caracteres
                               // recién extraídos
```

Gracias a las referencias ya no hace falta emplear punteros como ocurría en la función *scanf()* de la biblioteca de C.

El método `get(char* s, streamsize n, char d = '\n')` lee como mucho $n-1$ caracteres y los guarda en la zona apuntada por s, hasta encontrar el fin de la entrada o el carácter d, que hace de delimitador y que, como se ve, si no se suministra es el carácter nueva-línea. Este carácter delimitador no se guarda, y se añade siempre al final un carácter nulo `'\0'`.

El método *getline()* funciona como el anterior, sólo que no deja el carácter delimitador en el flujo de entrada; como si lo hubiera leído.

El método *read()* introduce como mucho n caracteres en la zona apuntada por s, pero si antes encuentra el fin de la entrada lo considera como error, activando *failbit* (§6.8).

Ejemplo:

```
int x, y;
char c, s[80];
cin.get(c);            // coge un carácter
cin.get(s, 40);        // cadena de 39 cars. o hasta EOF
cin.get(s, 9, ':');    // cadena hasta 8 caracteres o hasta ':'
cin.getline(s, 60);    // como get, pero '\n' se lee y desecha
cin >> x >> y;         // lee x e y
cin >> s;              // lee una cadena, delimitada por blancos
```

Note que en el último ejemplo no se lee una línea, sino caracteres delimitados por blancos (espacio, tabulador, nueva línea, etc.).

6.4.1. El operador de extracción >>

El operador >> se salta los espacios en blanco, por tanto se podría leer una secuencia de enteros separados por espacios en blancos de la siguiente forma:

```
vector<int> v(10);

for (vector<int>::size_type i = 0; i < v.size() && cin; ++i)
  cin >> v[i];
```

También se puede leer una cadena de bajo nivel directamente:

```
char v[4];

cin >> v;
cout << "v = " << v << endl;
```

El operador >> salta primero los espacios en blanco. A continuación lee a su operando vector hasta que encuentra un carácter de espacio en blanco o llega al final del fichero. Por último, termina la cadena con un 0. Esto ofrece claramente oportunidades de desbordamiento, por lo que habitualmente es mejor escribir en un *string*:

```
string s;
cin >> s;
```

Sin embargo, se puede especificar un máximo para el número de caracteres que va a leer >>. Así, `cin.width(n)` especifica que el >> siguiente en *cin* va a leer como máximo $n - 1$ caracteres en un vector.

Ejemplo:

ancho.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    char v[4];
7
8    cin.width(4);
9    cin >> v;
10   cout << "v = " << v << endl;
11 }
```

Esto leerá un máximo de tres caracteres en v y añadirá un 0 como terminador.

Especificar *width()* para un *istream* afecta sólo al `>>` que sigue inmediatamente a un vector y no afecta a otras lecturas.

Ahora se verá un programa completo que lee una secuencia de palabras de la entrada estándar y determina cuál es la mayor. El término *palabra* no hay que tomarlo al pie de la letra; sería correcto si sólo se suministraran al programa palabras separadas por blancos, sin signos de puntuación. Note la condición del bucle `while`; existe un operador definido en la clase que transforma *istream* a *void** (§6.8), y el puntero se transforma a *bool* en la condición automáticamente. Cuando se llega al final de la entrada la expresión da cero; o sea, falso.

pml.cpp

```
// Lee palabras de la entrada estándar,
// determina la más larga

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

int main()
{
  string palabra;
  string palabra_mas_larga;
  size_t longitud, max = 0, n = 0;  // estadísticas

  while (cin >> palabra) {
    longitud = palabra.length();
    ++n;
    // Si es la más larga, guardarla
    if (longitud > max) {
      max = longitud;
      palabra_mas_larga = palabra;
    }
  }
  cout << "El número total de palabras leídas es  "
       << n << endl
       << "La palabra más larga tiene de longitud "
       << max << endl
       << "La palabra más larga ha resultado ser  "
       << palabra_mas_larga << endl;
}
```

6.4.2. Entrada de caracteres

El operador >> está pensado para leer objetos de un tipo y un formato esperados. Cuando esto no es deseable y se quiere leer los caracteres como tales y luego examinarlos, se utiliza las funciones *get()* y *getline()*.

Los dos siguientes programas leen de la entrada estándar copiándola en la salida, como el conocido programa de UNIX `cat` empleado sin parámetros. Se llama a la función *get()* con un parámetro, una referencia a carácter. Como devuelve el objeto del mismo flujo, se puede utilizar como condición del bucle `while`, por lo dicho anteriormente de la conversión de *istream* a *void**.

cat1.cpp

```
1  // Copia la entrada a la salida (estándares)
2  // Ejemplo de get(char&)
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9    char c;
10   while (cin.get(c))
11     cout.put(c);
12 }
```

En este otro se llama sin parámetros, y entonces se parece a *getchar()*: al llegar al final de la entrada devuelve la constante *EOF*, definida en `<iostream>`.

cat2.cpp

```
1  // Copia la entrada a la salida (estándares)
2  // Ejemplo de get()
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9    int c;  // no char: para poder recibir EOF
10   while ((c = cin.get()) != EOF)
11     cout.put(c);
12 }
```

Por último, el siguiente programa, que lee un texto de la entrada estándar y lo imprime «en detalle», emplea *getline()*, *gcount()* y *write()*.

lineas.cpp

```
// Lee líneas de texto de la entrada estándar.
// Las copia en la salida indicando el número
// de caracteres de cada una; y, al final, la
// longitud de la más larga.

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

const int tambuf = 256;

int main()
{
  int cl  =  0;        // Cuántas líneas llevamos leídas
  int max = -1;        // Longitud de la mayor
  char cadena[tambuf]; // Para almacenar cada línea

  // Lee tambuf caracteres o hasta nueva-línea
  while (cin.getline(cadena, tambuf)) {
    // cuántos caracteres hemos leído de verdad
    int leidos = cin.gcount();
    // datos: número de línea, línea más larga...
    cl++;
    if (leidos > max)
      max = leidos;
    cout << "Línea número " << cl
         << "\tCaracteres leídos: " << leidos << endl;
    cout.write(cadena, leidos).put('\n').put('\n');
  }
  cout << "Total de líneas leídas: " << cl << endl
       << "Longitud de la línea más larga: " << max << endl;
}
```

6.4.3. Entrada de tipos definidos por el usuario

Como con el operador de inserción, el de extracción se puede sobrecargar en una clase para que ésta pueda recibir valores de una manera natural. El esqueleto sería así:

```
istream& operator >>(istream& is, T& obj)
{
  // Lógica para tratar con el objeto de tipo T
  return is;  // devolución del flujo de entrada
}
```

Ejemplo:

Retomando el ejemplo de los números racionales, ésta podría ser una forma muy abreviada y simplificada de introducción de un racional.

```
inline istream& operator >>(istream& is, Racional& r)
{
  is >> r.n;
  char c = 0;
  is >> c;
  if (c == '/')
    is >> r.d;
  else
    is.putback(c).clear(ios_base::badbit);
  return is;
}
```

6.5. Funciones virtuales de E/S

Los miembros de *iostream* no son `virtual`. Las operaciones de salida que se pueden definir no son miembros de ninguna clase, por lo que tampoco pueden ser `virtual`. La razón es lograr un mayor rendimiento en operaciones sencillas como poner un carácter en un búfer.

Sin embargo, a veces se quiere mandar a una salida un objeto y se quiere que se elija en tiempo de ejecución la definición del operador adecuada dentro de la jerarquía de clases. Esto no se puede hacer utilizándolo directamente. En su lugar se puede proporcionar en la clase base una función virtual de salida.

Ejemplo:

salida-virtual.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  class Base {
5  public:
6    // ...
7    virtual ostream& poner(ostream& s) const = 0;
8  };
9
10 class Derivada: public Base {
```

```
11  public:
12    // ...
13    ostream& poner(ostream& s) const { return s << d << endl; }
14  private:
15    int d;
16  };
17
18  ostream& operator <<(ostream& s, const Base& b)
19  {
20    return b.poner(s);
21  }
22
23  int main()
24  {
25    Base *b;
26    Derivada d;
27
28    b = &d;
29
30    cout << *b << d;
31  }
32
```

6.6. Flujos de fichero

Para la E/S de ficheros hay que incluir otra cabecera: `<fstream>`. En ella se definen las clases *ifstream* y *ofstream* para la creación y manipulación de ficheros de entrada y de salida respectivamente.

6.6.1. Apertura

Como ya se sabe, la *apertura* de un fichero es la operación que lo hace accesible al programa; se puede decir que se conecta un fichero (con posible creación si no existía, o truncamiento y vaciado) con un flujo que se manipula en el programa.

En la biblioteca de C existía una función para este propósito: *fopen()*. En la de C++ la apertura puede hacerse de dos maneras. De una o de otra, lo primero es definir un objeto del tipo *ifstream* u *ofstream*. Constructores:

```
ifstream();
ifstream(const char* camino,
         openmode modo = ios_base::in | ios_base::trunc);
```

Modo	Significado
`ios_base::in`	Lectura (entrada)
`ios_base::out`	Escritura (salida)
`ios_base::ate`	Colocación al final del fichero
`ios_base::app`	Modo de anexado; toda escritura ocurre al final del fichero
`ios_base::trunc`	Borra el contenido del fichero al abrir si ya existe
`ios_base::bin`	El fichero es binario; en UNIX no tiene efecto

Cuadro 6.14: Modos de apertura de ficheros

```
ofstream();
ofstream(const char* camino,
         openmode modo = ios_base::out | ios_base::trunc);
```

Si se ha empleado el constructor sin parámetros, entonces aún no se ha abierto el fichero, sólo se ha creado el objeto de tipo flujo; se tendrá que abrir luego mediante los métodos siguientes:

```
void
ifstream::open(const char* camino,
               openmode modo = ios_base::in | ios_base::trunc);
void
ofstream::open(const char* camino,
               openmode modo = ios_base::out | ios_base::trunc);
```

Tanto para las funciones *open()* como para los constructores con dos parámetros, sus significados son como sigue:

camino El nombre o camino del fichero que se quiere abrir. Los detalles dependen del sistema operativo, evidentemente. Si debe existir previamente o no, depende del modo de apertura.

modo El modo de apertura. Es un parámetro opcional. Los valores permitidos son constantes enumeradas de tipo *ios_base::open_mode*, como se muestran en la tabla 6.14. Se pueden especificar varios, aplicando el operador de bits '`|`'.

Si se define un objeto de la clase *fstream* el constructor hace que la apertura sea tanto para entrada como para salida; el parámetro *modo* debe ser suministrado por el usuario.

6.6.2. Cierre

El cierre de un fichero implica la desconexión del fichero físico con el flujo que se utiliza en el programa: el fichero ya no está accesible, los búferes se han vaciado. Existen tres formas (normales) de cerrar un fichero:

- Mediante el destructor de la clase asociada:

```
~ifstream();
~ofstream();
~fstream();
```

- Mediante el método *close()*.
- Mediante la función *exit()* de la biblioteca estándar de C (declarada en la cabecera `<cstdlib>`), que provoca la terminación normal del programa, o la instrucción `return` en la función *main()* o en definitiva una salida normal del programa (se llamaría al destructor).

Después de todo esto se van a ver algunos ejemplos.

Ejemplo:

Esto es un pequeño programa que recibe dos parámetros que son nombres de ficheros. Copia el primero en el segundo pero separando cada línea por una en blanco.

dblspc.cpp

```
1  // Copia un fichero a doble espacio
2  // Modo de empleo: dblspc <fichero-entrada> <fichero-salida>
3  // 'fichero-entrada' debe existir y poderse leer
4  // 'fichero-salida' debe poder ser modificado si existe
5
6  #include <fstream>  // ya incluye <iostream>
7  #include <cstdlib>  // para exit() y sus macros
8  using namespace std;
9
10 void doblespacia(ifstream& entrada, ofstream& salida);
11
12 int main(int argc, char* argv[])
13 {
14   if (argc != 3) {
15     cerr << "Modo de empleo: " << argv[0]
16          << " fichen fichsal\n";
17     exit(EXIT_FAILURE);
18   }
19
```

```
20    ifstream flujoent(argv[1]);
21    ofstream flujosal(argv[2]);
22    if (!flujoent) {
23      cerr << "No puedo abrir " << argv[1] << endl;
24      exit(EXIT_FAILURE);
25    }
26    if (!flujosal) {
27      cerr << "No puedo abrir " << argv[2] << endl;
28      exit(EXIT_FAILURE);
29    }
30    doblespacia(flujoent, flujosal);
31    return EXIT_SUCCESS;
32  }
33
34  void doblespacia(ifstream& e, ofstream& s)
35  {
36    char c;
37
38    while (e.get(c)) {
39      s.put(c);
40      if (c == '\n')
41        s.put(c);
42    }
43  }
```

Ejemplo:

Este otro programa imprime un fichero completo en la salida. Para ello hace uso del método *rdbuf()*, que devuelve un puntero al búfer interno del flujo. El *operator* << está sobrecargado para ese puntero, y muestra en el flujo especificado el búfer entero (el fichero en definitiva).

rdbuf.cpp

```
1   #include <iostream>
2   #include <fstream>
3   using namespace std;
4
5   int main()
6   {
7     ifstream motd("/etc/motd");
8
9     cout << motd.rdbuf();
10  }
```

`ios_base::beg`	relativo al principio
`ios_base::cur`	relativo a la posición actual
`ios_base::end`	relativo al final

Cuadro 6.15: Valores de la enumeración *seek_dir*

De forma similar se puede hacer un programa que copia la entrada a la salida.

entrada-salida.cpp

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
4
5 int main()
6 {
7   cout << cin.rdbuf();
8 }
```

6.6.3. Acceso directo

Si un flujo está conectado a un fichero en un dispositivo que permite acceso directo, es posible moverse a un punto determinado del fichero. Para ello se dispone de algunos métodos.

Los dos siguientes métodos permiten situarnos en la posición especificada por *desplto* a partir de lo indicado por el segundo parámetro, *orig*. Este segundo parámetro, el *origen* del *desplazamiento*, es opcional y puede tomar alguno de los valores mostrados en la tabla 6.15.

```
ostream& seekp(off_type desplto, ios_base::seekdir orig);
istream& seekg(off_type desplto, ios_base::seekdir orig);
```

Los dos siguientes permiten situarnos en una posición absoluta.

```
ostream& seekp(pos_type);
istream& seekg(pos_type);
```

Y los siguientes devuelven la posición actual en el flujo de entrada o salida respectivamente.

```
pos_type tellp();
pos_type tellg();
```

La última letra de estas funciones es *g*, por *get*, coger de la entrada, o *p*, por *put*, poner en la salida.

6.7. Flujos de cadena

En la biblioteca estándar de C existen las funciones *sprintf()* y *sscanf()* que leen y escriben no en un flujo sino en una cadena de caracteres que tratan como tal. De esta forma, se puede hacer conversiones de datos numéricos a o desde cadenas, aprovechando las capacidades de formato de dichas funciones, y guardar los resultados para su posterior procesado. La biblioteca *IOStreams* de C++ también permite esto.

Así, se puede conectar un flujo a un *string*. Es decir, se puede leer y escribir un *string* usando los recursos de formato que proporcionan los flujos. Estos flujos de cadena se denominan *stringstream* y se definen en `<sstream>`.

Por ejemplo, se puede utilizar un *ostringstream* para formatear objetos *string*.

```
string error(int n, const& string s)
{
  ostringstream o;

  o << "Error " << n << ": " << s << endl;
  return o.str();
}
```

No es necesario comprobar el desbordamiento porque *o* se expande a medida de las necesidades.

Se puede proporcionar un valor inicial para un *ostringstream*, por lo que se podría haber escrito de manera equivalente:

```
string error(int n, const string& s)
{
  ostringstream o("Error");

  o << n << ": " << s << endl;
  return o.str();
}
```

Un *istringstream* es un flujo de entrada que lee de un *string*:

```
void palabras(const string& s)
{
  istringstream i(s);
```

```
  string w;

  while (i >> w)       // lee una "palabra"
    cout << w << endl;
}
```

Ejemplo:

El siguiente programa lee de un fichero de configuración, `colores.ini`, nombres de colores y sus componentes RGB (rojo, verde, azul), y los muestra por la salida estándar formateados.

colores.cpp

```
1  // Lee del fichero colores.ini líneas en el formato
2  // color R G B
3  // y las imprime en la salida estándar con el formato
4  // color:        (R,G,B)
5  // con los números de los componentes en hexadecimal.
6
7
8  #include <fstream>
9  #include <iomanip>
10 #include <sstream>
11 #include <string>
12 using namespace std;
13
14 int main()
15 {
16   string color;     // nombre de color
17   unsigned int rojo, verde, azul;
18
19   // Abre el fichero de configuración para lectura
20   ifstream fc("colores.ini");
21
22   if (!fc) {
23     cerr << "Error al abrir colores.ini" << endl;
24     return 1;
25   }
26
27   string bufer;  // para leer cada línea
28   cout.setf(ios_base::uppercase); // mayúsculas
29   // Bucle de lectura línea a línea
30   while (getline(fc, bufer)) {
31     // Creamos un flujo en memoria sobre bufer
32     istringstream linea(bufer);
33     // Leemos «color» y los componentes de «linea»
34     linea >> color >> rojo >> verde >> azul;
```

```
35      cout << color << ":\t(" << hex
36           << rojo << "," << verde << ","
37           << azul << ")" << endl;
38    }
39  }
```

6.8. Empleo de los estados de los flujos

Cada flujo tiene asociado un estado que cambia en cada operación y que puede ser examinado para ver si ésta ha resultado bien o no. El estado de un flujo se representa como un conjunto de indicadores. Como la mayor parte de las constantes utilizadas para expresar el comportamiento de los flujos, esos indicadores se definen en la clase base *ios_base* de *basic_ios*:

```
class ios_base {
public:
  // ...
  static const iostate badbit,    // el flujo está corrompido
                       eofbit,    // fin de archivo
                       failbit,   // la operación sig. va a fallar
                       goodbit,   // goodbit == 0
  // ...
};
```

El estado de flujo se encuentra en la clase base *basic_ios* de *basic_stream* en `<ios>`:

```
bool good() const;        // la operación siguiente podría tener
                          //  éxito
bool eof() const;         // se ha llegado al final de la entrada
bool fail() const;        // la operación siguiente fallará
bool bad() const;         // el flujo está corrompido
iostate rdstate() const;  // devuelve los indicadores de estado
void clear(iostate f = goodbit);
                          // establece indicadores de estado
void setstate(iostate f) { clear (rdstate() | f); }
                          // suma f al estado
bool operator !() const;  // no good()
operator void*() const;   // distinto de cero si good()
```

Los significados de los estados son los siguientes:

good() La operación precedente de E/S funcionó bien, y la siguiente debería hacerlo.

eof() La operación precedente devolvió una condición de fin de fichero: se acabaron los datos de entrada, no había más que leer.

fail() La operación precedente es inválida. Pero si el estado no es también *bad()*, entonces aún se puede utilizar el flujo si la condición de error es corregida.

bad() La operación precedente falló y el flujo está corrompido. No hay ninguna posibilidad de recuperarlo.

Los dos operadores sirven para comprobar un flujo directamente. El operador *operator* `void *` es en realidad un método de conversión que convierte un flujo en un puntero genérico. Podría estar hecho así:

```
ios_base::operator void*()
{
  return fail() ? 0 : static_cast<void*>(this);
}

// Ejemplo de empleo

if (cin >> c)

  // ...procesar c; lectura OK

else

 // ...fallo de lectura
```

El operador *operator* `!` hace lo contrario, pero convirtiendo a `bool` para poder utilizarlo en una comprobación. Estaría hecho así:

```
bool ios_base::operator !() { return fail(); }

// Ejemplo de empleo:

cin >> a;
if (!cin) cerr "Error leyendo 'a'" << endl;
```

El siguiente programa ejemplo cuenta el número de «palabras» de la entrada estándar. De nuevo se entiende por «palabra» un conjunto consecutivo de caracteres no blancos; normalmente el programa se empleará redirigiendo la entrada estándar desde un fichero de texto.

cntplbrs.cpp

```
1  // Modo de empleo: cntplbrs < fichero
2
3  #include <iostream>
4  #include <cctype>    // Por isspace()
5  using namespace std;
6
7  int main()
8  {
9    int contador_palabras = 0;
10   int sgute_palabra();
11
12   while (sgute_palabra())
13     ++contador_palabras;
14   cout << "Número de \"palabras\": "
15        << contador_palabras << endl;
16 }
17
18 // Lee caracteres que forman una palabra
19 // Devuelve la longitud de la palabra
20 int sgute_palabra()
21 {
22   char c;
23   int longitud_palabra;
24
25   cin >> c;
26   for (longitud_palabra = 0;
27        !cin.eof() && !isspace(c);
28        ++longitud_palabra)
29     cin.get(c);
30   return longitud_palabra;
31 }
```

6.9. Mezcla de las bibliotecas de C y de C++

A pesar de todo lo visto puede que a un programador en C le resulten más cómodas, por estar ya habituado, algunas funciones de la biblioteca estándar de C. No hay nada malo en ello, ya se han mencionado las ventajas y desventajas de cada una al principio. Pero si se está tentado de mezclar funciones de la biblioteca de C con las de `<iostream>` puede haber problemas de sincronización debido a que usen distintas estrategias de manejo de búferes[5]. Esto puede evitarse con el método *ios_base::sync_with_stdio()* (el nombre es largo pero descriptivo).

[5] Tamponamiento, *bufering*.

Ejemplo:

esmix.cpp

```
1  // Mezclando las bibliotecas de E/S de C y C++
2
3  #include <cstdio>
4  #include <iostream>
5  using namespace std;
6
7  unsigned long int factorial(int n);
8
9  int main()
10 {
11   int n;
12
13    ios_base::sync_with_stdio();
14
15   do {
16     cout << "\nIntroduzca un número positivo n "
17          << "ó 0 para parar: ";
18     scanf("%d", &n);
19     printf("\n%d! = %lu", n, factorial(n));
20   } while (n > 0);
21   cout << "\nFin de la sesión" << endl;
22   return 0;
23 }
24
25 unsigned long int factorial(int n)
26 { // versión iterativa
27   unsigned long int f = 1ul;
28
29   for (int i = 2; i <= n; ++i)
30     f *= i;
31   return f;
32 }
33
```

6.10. Conclusión

La biblioteca de E/S de C++, llamada *IOStreams*, proporciona una forma segura y fiable de utilizar E/S mediante los *flujos*. Es preferible a la de C, llamada *stdio*[6], por la mejor comprobación de tipos. Para la lectura/escritura

[6] No se confunda: el fichero de cabecera `<cstdio>` es eso, un fichero de cabecera, no una biblioteca de funciones.

de clases proporciona métodos *naturales* de lectura/impresión mediante la sobrecarga de los operadores de inserción y extracción.

Los programadores con experiencia en C se resisten al uso de *IOStreams* y sólo la utilizan para lo más básico. Vale la pena dedicarle un poco de tiempo y hacer buen uso de ella porque tiene mucho que ofrecer y, cuando uno se acostumbra un poco, su empleo es más natural y claro. Compare por ejemplo la impresión de un número:

```
printf("%d\n", numero);
cout << numero << endl;
```

y compare la impresión de una variable de un TAD; por ejemplo, un complejo:

```
printf("%g%+gi", z.re, z.im);
cout << z;        // una vez sobrecargado <<, claro
```

Ejercicios

E6.1. El siguiente programa, en lugar de mostrar la dirección de memoria almacenada en el puntero *pstr*, imprime:

```
La dirección de pstr es: C/Chile s/n
```

Arregle el programa para que imprima lo que se espera.

malo1.cpp

```
1  // No funciona muy bien que digamos
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8    char str[] = "C/Chile s/n";
9    char* pstr = str;
10   cout << "La dirección de pstr es: "
11        << pstr << endl;
12 }
13
```

E6.2. El siguiente programa intenta mostrar el mayor de dos valores, pero el resultado es:

```
El mayor de (10, 20) es 0
```

¿Por qué ocurre eso? Arregle el programa.

malo2.cpp

```
1  // No funciona muy bien, que digamos
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8    int ix = 10, jx = 20;
9
10   cout << "El mayor de (" << ix << ", " << jx << ") es: ";
11   cout << (ix > jx) ? ix : jx;
12   cout << endl;
13 }
14
```

E6.3. Considere flujos de texto con datos sobre la estatura de personas compuestos por líneas con el siguiente formato:

```
nombre apellido estatura
```

donde el nombre y el apellido son cadenas de una única palabra, la estatura es un valor en coma flotante y los campos aparecen separados por espacios en blanco.

Escriba una clase *Persona* adecuada con operadores de inserción y extracción capaces de manejar flujos con este formato.

A continuación, escriba una función, llamada *persona_mas_alta()*, capaz de hallar la persona más alta de un flujo de estas características. En caso de existir dos personas con la misma estatura devolverá la primera de ellas.

Por último realice un programa de prueba que abra un fichero llamado `estaturas.dat` con el formato antedicho y muestre la (primera) persona más alta que contenga.

E6.4. ¿Qué cree que imprimirán las líneas siguientes?

a) `cout << setfill('0') << setw(5) << 12 << endl;`

b) `cout << 12 << oct << endl;`

c) `cout << hex << 12 << endl;`

E6.5. Escriba un pequeño programa que copie la entrada estándar a la salida estándar utilizando la función *rdbuf()*.

E6.6. Escriba un pequeño programa que copie la entrada estándar a la salida estándar expandiendo los tabuladores al número de blancos que indique su primer parámetro. Si el programa no recibe exactamente un parámetro debe mostrar un error y acabar.

E6.7. Sobrecargue paramétricamente el operador de extracción para objetos de tipo *vector*. El formato de entrada constará de:

- El carácter `'['`
- Una secuencia no vacía de elementos del tipo parámetro separados por comas, `','`.
- El carácter `']'`.

Cada uno de los elementos de formato puede ir precedido de un número arbitrario de blancos. Si se produce un error de formato, el operador debe modificar el estado del flujo para reflejar este hecho. En tal caso, el operador no debe modificar el objeto recibido como segundo parámetro, ni consumir caracteres del flujo que no correspondan a un prefijo de una entrada válida.

Capítulo 7

La Biblioteca Estándar de Plantillas (STL)

7.1. Introducción

La biblioteca estándar de plantillas (en inglés, *Standard Template Library* o STL) es una biblioteca de clases contenedoras, iteradores y algoritmos. Esta biblioteca proporciona implementaciones muy eficientes de gran parte de las estructuras de datos y algoritmos que se necesitan habitualmente.

Además, STL es una biblioteca diseñada para la programación genérica, lo que se refleja en el hecho de que prácticamente todos sus componentes son paramétricos.

Pero, sin duda, una de las características más sorprendentes e innovadoras de la STL es que como parte de su especificación incluye la complejidad asintótica temporal[1] de sus algoritmos. Esto significa que cualquier fabricante de compiladores o bibliotecas que desee cumplir el estándar de C++ está obligado a proporcionar implementaciones de sus algoritmos que cumplan ciertos requisitos de eficiencia.

Por ejemplo, las operaciones aplicables a un iterador (que veremos posteriormente) deben tardar un tiempo acotado superiormente por una constante independiente del tamaño del contenedor sobre el que actúe. Es decir, los tiempos de las operaciones de iteración pertenecen a $O(1)$ o, de otro modo, son constantes en el peor caso.

En ocasiones, la complejidad temporal se especifica mediante un *análisis amortizado*. Esto es útil cuando el tiempo de una operación puede sufrir

[1] Curiosamente, en prácticamente ningún lugar se imponen restricciones al espacio que puede consumir una operación.

grandes variaciones a lo largo de una secuencia de operaciones. En este caso un análisis en el peor caso podría ser excesivamente pesimista y un análisis en el promedio, poco significativo, por la gran desviación de los tiempos.

El tiempo amortizado de un algoritmo se define como el cociente entre el tiempo total de una secuencia de ejecuciones del algoritmo y la longitud de dicha secuencia. Su empleo es apropiado cuando el tiempo de una secuencia tal es notablemente inferior al producto de su longitud por el tiempo en el peor caso del algoritmo.

Por ejemplo, el tiempo en el peor caso[2] para insertar un elemento al final de un vector extensible (*vector*) pertenece a $O(n)$, siendo n la longitud del vector, ya que si no queda espacio adicional sin utilizar, la operación de inserción tendrá que reservar un nuevo espacio de memoria y mover todos los elementos a la nueva zona. Sin embargo, cuando hay que expandir un vector de longitud n no se reserva espacio para un único elemento adicional, sino para $2n$. Esto permite que las siguientes $n-1$ inserciones no necesiten expandir el vector.

Esto implica que las $n-1$ inserciones posteriores tienen tiempos constantes, con lo que el tiempo total de una secuencia de n inserciones pertenece a $O(n)$. Amortizando este tiempo sobre las n inserciones se obtiene $O(1)$; se dice, pues, que la inserción al final de un vector es una operación de *tiempo amortizado constante*.

En adelante, para abreviar, emplearemos O_A cuando nos refiramos a un orden asintótico proveniente de un análisis amortizado. También emplearemos n para designar el número de elementos que intervienen en una operación, es decir, el tamaño de la entrada.

7.2. Contenedores e iteradores

7.2.1. Contenedores

Un *contenedor* es un objeto que se utiliza para almacenar otros objetos proporcionando operaciones para su manipulación. Como se verá, STL posee diversas clases contenedoras agrupadas en dos categorías: secuencias y contenedores asociativos ordenados. Los nombres de las clases y sus cabeceras estándares aparecen en la tabla 7.1.

Todos los contenedores poseen una serie de características comunes:

1. Definiciones públicas de tipos (véase la tabla 7.2).

[2] Contado en número de asignaciones de elementos.

Nombre	Cabecera	Descripción
vector	`<vector>`	Vectores
deque	`<deque>`	Colas dobles
list	`<list>`	Listas
set, *multiset*	`<set>`	Conjuntos y multiconjuntos
map, *multimap*	`<map>`	Aplicaciones mono y multivalor

Cuadro 7.1: Contenedores de la STL

2. Constructor por omisión que construye un contenedor vacío.
3. Constructor de copia y operador de asignación definidos con la semántica habitual.
4. Destructor.
5. Funciones miembro de tamaño (véase la tabla 7.3).
6. Operador `==` de igualdad estructural y operador `<` de comparación lexicográfica estricta.
7. Operadores relacionales adicionales (`!=`, `>`, `<=` y `>=`), definidos con la semántica habitual en función de `==` y `<`.
8. Funciones miembro de construcción de iteradores (véase la tabla 7.4).
9. Función miembro *swap()* que intercambia un contenedor por otro (normalmente en tiempo constante).

7.2.2. Iteradores

Los contenedores proporcionan *iteradores* para que a través de éstos sea posible recorrer sus elementos en un orden prefijado. Un iterador es una abstracción del concepto de puntero; de hecho, los punteros pueden emplearse en cualquier contexto en el que se requiera un iterador.

Los iteradores permiten separar los contenedores de los algoritmos genéricos. STL proporciona un rico conjunto de algoritmos genéricos que no trabajan directamente con los contenedores, sino con los iteradores proporcionados por dichos contenedores. Esto hace posible lo siguiente:

Tipo	Descripción
value_type	Tipo de los elementos del contenedor
reference	Tipo apropiado para almacenar elementos en el contenedor.
const_reference	Tipo apropiado para almacenar elementos constantes en el contenedor.
pointer	Tipo apropiado para apuntar a elementos del contenedor.
const_pointer	Tipo apropiado para apuntar a elementos constantes del contenedor.
iterator	Tipo apropiado para recorrer los elementos del contenedor.
const_iterator	Tipo apropiado para recorrer los elementos del contenedor, si ha sido definido constante.
reverse_iterator	Tipo apropiado para recorrer los elementos del contenedor en orden inverso.
const_reverse_iterator	Tipo apropiado para recorrer los elementos del contenedor en orden inverso, si ha sido definido constante.
difference_type	Tipo entero con signo, apropiado para representar la diferencia entre dos iteradores.
size_type	Tipo entero sin signo, apropiado para representar el tamaño del contenedor.

Cuadro 7.2: Miembros de tipo

Nombre	Complejidad	Descripción
empty()	$O_A(1)$	¿Está vacío el contenedor?
size()	$O(n)$	Tamaño (número de elementos) del contenedor.
max_size()	$O_A(1)$	Cota superior del tamaño del contenedor.

Cuadro 7.3: Funciones miembro de tamaño

Nombre	Complejidad	Descripción
begin()	$O_A(1)$	Devuelve un iterador apropiado para comenzar a recorrer el contenedor.
end()	$O_A(1)$	Devuelve un iterador apropiado para comprobar el final de un recorrido.
rbegin()	$O_A(1)$	Devuelve un iterador apropiado para comenzar a recorrer el contenedor en orden inverso.
rend()	$O_A(1)$	Devuelve un iterador apropiado para comprobar el final de un recorrido en orden inverso.

Cuadro 7.4: Funciones miembro de construcción de iteradores

1. Que un mismo algoritmo pueda aplicarse a distintos contenedores estándar, siempre que los iteradores que proporcionen sean «compatibles»[3] con los requeridos por el algoritmo.

2. Que un algoritmo pueda aplicarse a nuevos contenedores definidos por el usuario, siempre que éstos proporcionen iteradores que cumplan unos requisitos prefijados.

3. Que un algoritmo pueda aplicarse a tipos primitivos del lenguaje, puesto que los punteros son una especie de iteradores.

Las clases contenedoras permiten obtener iteradores adecuados a sus características particulares mediante las funciones de construcción de iteradores de la tabla 7.4.

Cada una de estas funciones de construcción de iteradores posee dos versiones, una normal y otra que es una sobrecarga `const` de ésta. Las funciones *begin()* y *end()* devuelven iteradores ordinarios (*iterator* o *const_iterator*) mientras que, en cambio, *rbegin()* y *rend()* devuelven iteradores inversos (*reverse_iterator* o *const_reverse_iterator*).

Dados dos iteradores *principio* y *fin*, se define su *rango* asociado, que se denota por $[principio, fin)$, como la secuencia de valores que comienza en *principio* y llega hasta *fin* sin incluirlo. Un rango cuyos extremos son iguales se denomina *rango vacío*.

[3]Esta compatibilidad se ha fijado dentro de STL atendiendo a razones de eficiencia; un algoritmo genérico no funcionará con un iterador del cual necesite operaciones cuya complejidad intrínseca aumente más allá de lo razonable la del propio algoritmo.

Las funciones *end()* y *rend()* siempre indican *una posición más allá* del elemento final del recorrido; por lo tanto, para recorrer un contenedor c completamente hay que iterar sobre el rango [`c.begin()`,`c.end()`).

Las clases y operaciones relacionadas con los iteradores se encuentran en `<iterator>`, que a su vez se incluye en muchos otros lugares.

Ejemplo:

Veamos diversas formas de recorrer un vector, por ejemplo, para mostrarlo en un flujo de salida. Para ello sobrecargaremos paramétricamente *operator* `<<` para que trabaje con vectores.

La primera consiste en acceder secuencialmente a cada uno de los elementos del vector mediante el operador de índice.

```
template <typename T>
ostream& operator <<(ostream& fs, const vector<T>& v)
{
  fs << "[ ";
  for (vector<T>::size_type i = 0; i < v.size(); ++i)
    fs << v[i] << ' ';
  return fs << ']';
}
```

Nótese cómo *vector<T>::size_type* es el tipo apropiado para representar un índice válido dentro del vector. De hecho, éste es el tipo devuelto por *vector<T>::size()*.

Pero existe otra opción: emplear iteradores. Para recorrer el vector completo hay que iterar sobre el rango [`v.begin()`,`v.end()`).

```
template <typename T>
ostream& operator <<(ostream& fs, const vector<T>& v)
{
  fs << "[ ";
  for (vector<T>::const_iterator i = v.begin(); i != v.end(); ++i)
    fs << *i << ' ';
  return fs << ']';
}
```

Observe que tanto *vector<T>::begin()* como *vector<T>::end()*, cuando se aplican a v, devuelven un *vector<T>::const_iterator*, ya que v es `const`.

Supongamos ahora que quisiéramos mostrar el vector «al revés». Al intentar modificar la versión con operador de índice tropezamos con un pequeño inconveniente, ya que

```
for (vector<T>::size_type i = v.size() - 1; i >= 0; --i)
  fs << v[i] << ' ';
```

es un bucle infinito. Aunque esto es fácil de arreglar de múltiples formas, la versión con iteradores se obtiene directamente empleando iteradores inversos:

```
for (vector<T>::const_reverse_iterator i = v.rbegin();
     i != v.rend(); ++i)
  fs << *i << ' ';
```

Es decir, para recorrer el vector completo «al revés» hay que iterar sobre el rango [`v.rbegin()`,`v.rend()`).

Además, las versiones con iteradores son ligeramente más eficientes en la práctica que las que emplean el operador de índice, por lo que se prefieren normalmente. Esto es similar a lo que ocurre en C, donde se puede recorrer un vector empleando punteros en vez del operador de índice (el cálculo de la dirección es, a nivel de máquina, algo más costoso que el incremento de un puntero).

Clasificación

Todos los tipos de iteradores definidos en STL disponen de sobrecargas de *operator* `*` y *operator* `++`, dividiéndose en diversas categorías dependiendo de las operaciones adicionales que se pueden realizar sobre ellos.

Iteradores de entrada Son aquéllos que permiten realizar un recorrido en una sola dirección leyendo el elemento apuntado. Sobrecargan también *operator* `->`.

Iteradores de salida Son aquéllos que permiten realizar un recorrido en una sola dirección escribiendo sobre el elemento apuntado.

Iteradores monodireccionales Son aquéllos que permiten realizar un recorrido en una sola dirección leyendo o escribiendo sobre el elemento apuntado. Sobrecargan también *operator* `->`.

Iteradores bidireccionales Son aquéllos que permiten realizar un recorrido en ambas direcciones leyendo o escribiendo sobre el elemento apuntado. Sobrecargan también *operator* `->` y *operator* `--`.

Iteradores de acceso directo Son aquéllos que permiten todas las operaciones que se pueden realizar con un puntero ordinario.

Todos los iteradores, a excepción de los de salida, poseen sobrecargas de *operator* `==` y *operator* `!=`.

Como se observa, los iteradores de entrada no garantizan que se pueda modificar el resultado de una operación `*` o `->`, mientras que los de salida sí garantizan la modificación a través de `*`, pero no la lectura.

Nótese también que los iteradores monodireccionales, bidireccionales y de acceso directo son *a la vez* iteradores de entrada y de salida.

Cada contenedor proporciona los iteradores apropiados para que puedan emplearse algoritmos eficientes sobre ellos. Los vectores y las colas dobles proporcionan iteradores de acceso directo[4]. Sin embargo, los iteradores de las listas y los contenedores asociativos son únicamente bidireccionales, ya que no es posible realizar acceso directo a un elemento de estos contenedores en tiempo constante[5].

Así, un algoritmo genérico diseñado eficientemente para emplear acceso directo, como es el caso del algoritmo de ordenación *sort()* de STL, sería ineficiente si se aplicara a una lista que proporcionara iteradores de «acceso directo» de complejidad $O(n)$. Por lo tanto, las listas no proporcionan tales iteradores y para compensar poseen una función miembro *sort()* especial.

Ejemplo:

El algoritmo genérico *copy()* recibe tres iteradores como parámetros. Los dos primeros son de entrada y el último es de salida. Esto permite el empleo de este algoritmo para transferir datos entre dos contenedores cualesquiera.

Iteradores de inserción

Si se escribe sobre un contenedor a través de un iterador de salida sin precaución, se estarán sobreescribiendo sus elementos. A menudo no es esto lo que se desea y suele ser causa de errores por desbordamiento del contenedor.

Ejemplo:

```
vector<double> v(10, 1.0), w(5, 2.0);
copy(v.begin(), v.end(), w.begin()); // Mal, «w» no tiene espacio
                                     // suficiente reservado
```

[4]El término «acceso directo» significa que se puede acceder a cualquier elemento en tiempo constante, es decir, que se puede considerar como una operación elemental.

[5]Dicho de otro modo, el acceso a un elemento no sería una operación elemental.

Más bien, lo que se desea en muchas ocasiones es algún tipo de inserción en el contenedor. Para lograr esto, STL define tres clases paramétricas denominadas *iteradores de inserción*; por cada una de éstas se define también una función de inserción o *insertor* que produce el iterador adecuado:

Insertores delanteros Se emplean para insertar elementos al principio de un contenedor que reciben como parámetro. Su nombre de función es *front_inserter()* y generan un objeto de tipo *front_insert_iterator*.

Insertores traseros Se emplean para insertar elementos al final de un contenedor que reciben como parámetro. Su nombre es *back_inserter()* y generan un *back_insert_iterator*.

Insertores «in situ» Se emplean para insertar elementos en un contenedor, que recibe como primer parámetro, en la posición anterior a la dada por un iterador, que recibe como segundo parámetro. Su nombre es *inserter()* y generan un *insert_iterator*.

Ejemplo:

```
vector<double> v(10, 1.0);
list<double> w(5, 2.0);
copy(v.begin(), v.end(), front_inserter(w)); // ins. al principio
copy(v.begin(), v.end(), back_inserter(w));  // ins. al final
```

Iteradores de flujo

STL proporciona clases que permiten tratar los flujos de E/S como si fueran secuencias. A estas clases se las denomina genéricamente *iteradores de flujo*.

La clase paramétrica *ostream_iterator* permite construir iteradores de salida capaces de escribir sobre un flujo de salida. Al construir un objeto de este tipo se le asocia un flujo de salida y, opcionalmente, una cadena de bajo nivel que actúa como terminador insertándose en el flujo automáticamente tras cada elemento.

Ejemplo:

Veamos otra forma de sobrecargar paramétricamente *operator* << para mostrar un vector en un flujo de salida.

```
template <typename T>
ostream& operator <<(ostream& fs, const vector<T>& v)
{
  fs << "[ ";
  copy(v.begin(), v.end(), ostream_iterator<T>(fs, " "));
  return fs << ']';
}
```

El objeto *ostream_iterator* permite insertar valores de tipo T separados por `" "` en el flujo de salida *fs*.

La clase paramétrica *istream_iterator* permite construir iteradores de entrada capaces de leer de un flujo de entrada. Al construir un objeto de este tipo se le puede asociar un flujo de entrada; no obstante, existe un constructor por omisión que crea un objeto apropiado para representar el fin de la entrada.

Ejemplo:

El siguiente fragmento lee números enteros de la entrada estándar separados por espacio blanco y los va insertando al final de un vector.

```
vector<int> v;
istream_iterator<int> ife(cin), // iterador de flujo de entrada
                      fin;      // fin de flujo de entrada
copy(ife, fin, back_inserter(v));
```

Observe cómo *fin* no lleva asociado ningún flujo concreto.

7.3. Secuencias

Una *secuencia* es un contenedor de elementos sobre los que no existe, a priori o de manera natural, un orden prefijado, y que presenta al exterior como si estuvieran dispuestos linealmente. Se definen tres tipos de secuencia: vectores, colas dobles y listas.

> Adicionalmente, los vectores de bajo nivel pueden ser considerados como secuencias en el sentido de que todos los algoritmos genéricos de STL que trabajan con secuencias lo hacen también con vectores de bajo nivel. Esto es así gracias a que los punteros funcionan, a todos los efectos, como iteradores de acceso directo.

Además de las características comunes a todas las clases contenedoras, las secuencias poseen:

Constructores por relleno

Reciben como primer parámetro un tamaño y como segundo (omitible) un valor apropiado para construir una secuencia de dicho tamaño con copias del valor. En caso de omitir el segundo parámetro se presupone el constructor por omisión del tipo de los valores.

Constructor por rango

Recibe dos iteradores de entrada apropiados y construye una secuencia que es una copia de los elementos de dicho rango.

assign()

Elimina todos los elementos de una secuencia sustituyéndolos por otros. Hay dos versiones: la primera es análoga al constructor por relleno y la segunda, al constructor por rango.

front()

Devuelve el primer elemento de la secuencia. La secuencia no debe estar vacía.

back()

Devuelve el último elemento de la secuencia. La secuencia no debe estar vacía.

resize()

Modifica el tamaño de la secuencia al valor indicado por su primer parámetro. Ya que puede implicar la creación de nuevos elementos, recibe como segundo parámetro (omitible) un valor apropiado para construirlos. En caso de omitir el segundo parámetro se presupone el constructor por omisión del tipo de los valores.

push_back()

Inserta un elemento al final de la secuencia.

insert()

Hay tres versiones, pero todas reciben como primer parámetro un iterador e insertan en la posición anterior a la indicada por él. La primera inserta un elemento, que recibe como segundo parámetro. La segunda generaliza a la anterior permitiendo la inserción de un cierto número de copias. Por último, la tercera permite insertar los elementos contenidos en un rango que recibe como segundo y tercer parámetros.

pop_ back()

Elimina el último elemento de la secuencia. La secuencia no debe estar vacía.

erase()

Existen dos versiones: una elimina el elemento dado por un iterador y otra los indicados por un rango. Ambas devuelven un iterador a la posición posterior al último elemento eliminado.

clear()

Elimina todos los elementos de la secuencia.

EJEMPLO:

```
vector<double> v(100, 1.0);            // construcción por relleno
list<double> l(v.begin(), v.end());    // construcción por rango
v.assign(100, 2.0);                    // asignación por relleno
l.assign(v.begin(), v.end());          // asignación por rango
v.resize(200);                         // expansión; relleno con 0.0
l.resize(200, 2.0);                    // expansión; relleno con 2.0
v.insert(v.end(), 300, 3.0);           // inserción al final
l.insert(l.end(), v.begin(), v.end()); // inserción al final
v.erase(v.begin(), v.end());           // vaciado
l.clear();                             // ídem, pero más corto
```

7.3.1. Vectores

Proporcionan acceso directo a una secuencia de longitud variable. Los vectores de STL son extensibles por el final, por lo que también permiten insertar y eliminar elementos al final eficientemente (tiempo amortizado constante).

Además de los miembros propios de una secuencia, un vector posee los siguientes:

capacity()

Devuelve la capacidad del vector, es decir, el número de elementos para los que se ha reservado espacio.

reserve()

Asegura que la capacidad del vector no sea inferior a un cierto valor, que recibe como parámetro; en caso de serlo, reserva espacio adicional para el número de elementos necesarios.

operator `[]`

Accede al elemento del vector indicado por su parámetro. El elemento debe existir.

at()

Análoga a *operator* `[]`, pero si el elemento no existe lanza una excepción *out_of_range*.

Nótese que la capacidad de un vector es, en general, distinta de su tamaño: no todas las posiciones reservadas tienen por qué estar ocupadas. Cuando durante la inserción de un elemento se sobrepasa la capacidad del vector, se reserva automáticamente espacio adicional.

> El espacio reservado cuando se sobrepasa la capacidad de un vector suele ser un múltiplo de ésta. Esto permite que las operaciones de inserción al final tengan tiempo amortizado constante.

La función miembro *reserve()* permite realizar esta reserva explícitamente. Téngase en cuenta que esta función afecta exclusivamente a la capacidad del vector, no a su tamaño.

Ejemplo:

La función *conversion()* del siguiente listado, calcula los dígitos de un número n en base b almacenándolos en un vector en orden creciente de pesos (el menos significativo en la primera, y así sucesivamente).

conversion/conversion.cpp

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <cassert>
5  using namespace std;
6
7  typedef unsigned int natural;
8
9  vector<natural> conversion(natural n, natural b);
10
11 int main()
12 {
13   natural n, b;
14   cout << "n = ";
15   cin >> n;
```

```
16    cout << "b = ";
17    cin >> b;
18    vector<natural> digitos = conversion(n, b);
19    copy(digitos.rbegin(), digitos.rend(),
20         ostream_iterator<natural>(cout, " "));
21    cout << endl;
22  }
23
24  vector<natural> conversion(natural n, natural b)
25  {
26    assert(b >= 2);
27    vector<natural> digitos;
28    do {
29      digitos.push_back(n % b);
30    } while (n /= b);
31    return digitos;
32  }
```

Nótese cómo el vector se expande automáticamente y no es necesario conocer por anticipado su tamaño máximo. Compárelo con lo que sería una función equivalente en C.

7.3.2. Colas dobles

Al igual que los vectores, proporcionan acceso directo a una secuencia de longitud variable, con la diferencia de que permiten insertar y eliminar elementos en tiempo amortizado constante tanto al final como al principio de la secuencia.

Además de los miembros propios de una secuencia, una cola doble posee los siguientes:

push_front()

Inserta un elemento al principio de la cola.

pop_front()

Elimina el primer elemento de la cola. La cola no debe estar vacía.

operator `[]`

Análogo al de los vectores.

at()

Análoga a la de los vectores.

7.3.3. Listas

A diferencia de vectores y colas, el acceso a los elementos en una lista no es directo, sino que su tiempo es lineal en el número de elementos que contiene. No obstante, se permite insertar y eliminar en tiempo constante en cualquier posición de la secuencia.

Además de los miembros propios de una secuencia, una lista posee principalmente los siguientes:

push_front()

Análoga a la de las colas dobles.

pop_front()

Análoga a la de las colas dobles.

remove()

Elimina todos los elementos de la lista iguales al que recibe como parámetro en n comparaciones. Trabaja con *operator* ==.

remove_if()

Elimina todos los elementos de la lista que cumplen un cierto predicado en n aplicaciones de éste. El predicado viene determinado por un objeto con la función de comparación que recibe como parámetro.

unique()

Elimina todos los elementos, salvo el primero, en cada secuencia consecutiva de elementos iguales de la lista en $n - 1$ comparaciones. Existen dos versiones: una que no recibe parámetros y trabaja con *operator* ==, y otra que recibe un objeto con la función de comparación.

merge()

Funde[6] la lista que recibe como parámetro sobre la que recibe implícitamente en no más de $n - 1$ comparaciones, siendo n el número total de elementos. Ambas deben estar previamente ordenadas; la primera queda ordenada y la otra vacía. Existen dos versiones: una que recibe un único parámetro y trabaja con *operator* <, y otra que recibe además un objeto con la función de comparación.

reverse()

Invierte el orden de la lista en tiempo lineal.

[6] La fusión es *estable*, es decir, se preserva el orden relativo de los elementos equivalentes.

sort()

Ordena[7] la lista en $\Theta(n \log n)$ comparaciones. Existen dos versiones: una que no recibe parámetros y trabaja con *operator* <, y otra que recibe un objeto con la función de comparación.

Ejemplo:

La siguiente función lee palabras (cadenas de caracteres separadas por espacio blanco) de un flujo de entrada almacenándolas en una lista. Cuando la entrada acaba ordena la lista obtenida con la función miembro *sort()* y la muestra por un flujo de salida.

ordenar-palabras/ordenar-palabras.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <list>
4  #include <algorithm>
5
6  // Ordena las palabras de un flujo y las escribe en otro
7
8  void ordenar_palabras(istream& fe, ostream& fs, bool repeticiones)
9  {
10   istream_iterator<string> entrada(fe), fin;
11   list<string> palabras(entrada, fin);
12   palabras.sort();
13   if (!repeticiones)
14     palabras.unique();
15   ostream_iterator<string> salida(fs, "\n");
16   copy(palabras.begin(), palabras.end(), salida);
17 }
```

El tercer parámetro de la función controla si se han de filtrar las cadenas repetidas. En tal caso, la función miembro *unique()* se encarga de su eliminación.

7.4. Adaptadores de secuencia

Un *adaptador* es un componente que modifica la interfaz de otro. Mientras que ninguna de las tres secuencias estudiadas (vectores, colas dobles y listas)

[7] La ordenación es *estable*, es decir, se preserva el orden relativo de los elementos equivalentes.

se puede construir eficientemente a partir de otra de ellas, se pueden definir pilas y colas simples eficientemente a partir de éstas.

En vez de definir las pilas y las colas simples como clases contenedoras ordinarias, STL las define como *adaptadores de secuencia*, es decir, como clases que adaptan la interfaz de las secuencias básicas a la de una pila o cola ordinaria.

Existen tres adaptadores de secuencia que nos permiten manejar pilas, colas simples y colas de prioridades. Sus nombres y cabeceras estándares aparecen en la tabla 7.5.

Nombre	Cabecera	Descripción
stack	`<stack>`	Pilas
queue	`<queue>`	Colas simples
priority_queue	`<priority_queue>`	Colas simples de prioridades

Cuadro 7.5: Adaptadores de secuencia de la STL

Estos adaptadores se comportan como «envoltorios» que restringen y renombran las operaciones que se pueden realizar sobre los contenedores; de hecho, ni siquiera poseen iteradores. Cada objeto adaptador tiene un contenedor subyacente que se indica como segundo parámetro de su plantilla; si no aparece, se toma un contenedor por omisión (colas dobles para las pilas y colas simples, vectores para las colas de prioridades).

Ejemplo:

```
stack<int, vector<int> > p1; // pila basada en un vector
stack<int, list<int> > p2;   // pila basada en una lista
stack<int, deque<int> > p3;  // pila basada en una cola doble
stack<int> p4;               // ídem (contenedor por omisión)
```

Cada uno de los adaptadores de secuencia define públicamente tres tipos: *container_type*, *value_type* y *size_type*. El primero es el tipo del contenedor de secuencia subyacente al adaptador; los otros dos tienen significados análogos a los de los contenedores estándar.

Todos comparten también las siguientes funciones miembro:

size()

Análoga a la de los contenedores ordinarios.

empty()

Análoga a la de los contenedores ordinarios.

push()

Introduce un elemento. La forma concreta de hacerlo depende del adaptador: para una pila significa apilar en la cima; para una cola, poner el último; para una cola con prioridades, poner en el orden apropiado según su prioridad.

pop()

Elimina un elemento. La forma concreta de hacerlo depende del adaptador: para una pila significa desapilar de la cima; para una cola, eliminar el primero; para una cola con prioridades, eliminar el de mayor prioridad.

Mientras que *push()* y *pop()* emplean tiempo amortizado constante para pilas y colas simples, requieren $O(\log n)$ para colas de prioridades.

Las pilas poseen funciones miembro *top()* para acceder a la cima; las colas simples, *front()* y *back()*; y las colas de prioridades poseen una función miembro *top()* para obtener el elemento de mayor prioridad.

Ejemplo:

En el siguiente listado se define una función, *equilibrada()*, que recibe una expresión representada por una cadena de caracteres y decide si está «equilibrada» respecto de un conjunto predefinido de símbolos de apertura y de cierre. En principio, se emplearán los símbolos de la siguiente tabla:

Nombre	Apertura	Cierre
Llaves	{	}
Corchetes	[	]
Paréntesis	(	)
Comillas	«	»
Interrogación	¿	?
Admiración	¡	!

La expresión estará equilibrada si todo símbolo de apertura se compensa posteriormente con el símbolo de cierre que le corresponde, y no con ningún otro símbolo de cierre.

Así, expresiones equilibradas serían:

```
{[(x + y) ^ 2 + 1] * [z + (-1) ^ n]}
«¡¡¡Qué bien, ¿no?!!!»
```

Mientras que las siguientes expresiones estarían desequilibradas:

```
{[(x + y) ^ 2 + 1  * [z + (-1) ^ n]}
«¡¡¡Qué bien, ¿no!!!?»
```

La función emplea una pila para almacenar los símbolos de apertura encontrados, en espera de poder cerrarlos y desapilarlos.

equilibrada/equilibrada.cpp

```
1  #include <string>
2  #include <stack>
3  using std::string;
4  using std::stack;
5
6  // Símbolos de apertura y de cierre
7  // Deben corresponderse uno a uno en idénticas posiciones
8
9  namespace simbolo {
10   const string apertura = "{[(«¿¡",
11                cierre   = "}])»?!";
12 }
13
14 // ¿Es un símbolo de apertura?
15
16 inline bool apertura(char c)
17 {
18   return simbolo::apertura.find(c) != string::npos;
19 }
20
21 // ¿Es un símbolo de cierre?
22
23 inline bool cierre(char c)
24 {
25   return simbolo::cierre.find(c) != string::npos;
26 }
27
28 // ¿Se compensa un símbolo de apertura con otro de cierre?
29
30 inline bool compensa(char c1, char c2)
31 {
32   return simbolo::apertura.find(c1) == simbolo::cierre.find(c2);
33 }
34
35 // ¿Está una expresión equilibrada?
```

```
36
37 bool equilibrada(const string& expr)
38 {
39   // Construimos una pila de caracteres vacía
40   stack<char> pila;
41   // Recorremos la expresión carácter a carácter
42   for (string::size_type i = 0; i < expr.size(); ++i) {
43     // Obtenemos el siguiente carácter de la expresión
44     char c1 = expr[i];
45     // Si es un símbolo de apertura, lo apilamos
46     if (apertura(c1))
47       pila.push(c1);
48     // Si es de cierre...
49     else if (cierre(c1)) {
50       // ...y la pila está vacía, es que dicho símbolo no
51       // se abrió, y la expresión está desequilibrada
52       if (pila.empty())
53         return false;
54       // ...y la pila no está vacía, obtenemos el símbolo
55       // de apertura que se encuentra en la cima de la pila
56       char c2 = pila.top();
57       // si no se compensa el símbolo de apertura con el de
58       // cierre, la expresión está desequilibrada
59       if (!compensa(c2, c1))
60         return false;
61       // y si se compensa, lo desapilamos
62       pila.pop();
63     }
64   }
65   // Si la pila está vacía, todos los símbolos de apertura han
66   // sido cerrados y la expresión está equilibrada; si no,
67   // queda alguno por cerrar y está desequilibrada
68   return pila.empty();
69 }
```

7.5. Contenedores asociativos (ordenados)

Las clases de la Biblioteca Estándar de Plantillas (STL) de C++ *set*, *map*, *multiset* y *multimap* se llaman *contenedores asociativos* porque asocian *claves* con *valores*. Esto es ciertamente verdad para *map* y *multimap*, pero un *set* puede verse como un *map* que no tiene valores, sólo claves, y lo mismo puede decirse de la relación entre *multiset* y *multimap*; de forma que debido a la similitud estructural, *set* y *multiset* se estudian junto con los contenedores asociativos.

Son además contenedores asociativos *ordenados* porque sus elementos se guardan en orden ascendente de sus claves. Para la comparación de dos elementos, el usuario puede suministrar un *predicado binario* u objeto función (se estudian más adelante) que compare dos objetos y devuelva `true` si el primero precede al segundo. Este predicado debe obedecer la definición matemática de un *orden débil estricto*; informalmente, debe comportarse como *menor que*: $a < b \Rightarrow b \not< a$ (propiedad antisimétrica), $a < b \wedge b < c \Rightarrow a < c$ (transitiva). De hecho, si no se suministra la comparación, se emplea por omisión el objeto función *less*, definido en `<functional>` así:

```
template <class T>
struct less: public binary_function<T, T, bool> {
  bool operator()(const T& x, const T& y) const { return x < y; }
};
```

Observe que utiliza el *operator* `<` para comparar los dos objetos de tipo T.

Por lo tanto podemos definir un objeto contenedor asociativo ordenado, por ejemplo un conjunto de enteros, así:

```
set<int> s1;                // se ordenarán con < de int's
set<int, less<int> > s2;    // igual, pero explícito
set<int, greater<int> > s3; // se ordenarán al revés
set<int, cmp<int> > s4;     // se ordenarán con nuestro «cmp»
```

> Más en detalle: nuestro predicado de comparación $f(x, y)$ debe satisfacer lo siguiente: debe devolver `true` si x precede a y, y `false` en caso contrario.
>
> Dos objetos x e y son *equivalentes* si ambos $f(x, y)$ y $f(y, x)$ dan *falso*.
>
> Las propiedades que debe cumplir f son:
>
> **antirreflexiva** $f(x, x)$ debe ser falso.
> **antisimétrica** $f(x, y) \Rightarrow \neg f(y, x)$
> **transitiva** $f(x, y) \wedge f(y, z) \Rightarrow f(x, z)$
>
> Estas propiedades definen un *orden parcial*. Un *orden débil estricto* debe además cumplir la propiedad *transitiva de la equivalencia*: si $x \equiv y \wedge y \equiv z \Rightarrow x \equiv z$.

De modo resumido, la complejidad de las operaciones de búsqueda es $O(\log n)$ y la de las operaciones de lista (esto es, inserciones y eliminaciones) es $O_A(\log n)$.

Las operaciones básicas más importantes sobre los contenedores asociativos son poner cosas en ellos y, en el caso de un *conjunto* (*set*), ver si algo está

en él (si un elemento pertenece al conjunto). En el caso de una *aplicación* (*map*), normalmente primero se quiere ver si una cierta clave está en ella, y si es así, recuperar su valor asociado.

Para el ejemplo a continuación se emplea la clase *Cosa*, y un par más de clases auxiliares, que sólo sirven para ejemplos y ejercicios, pues su única utilidad es mostrar cuándo ocurren las construcciones, asignaciones y destrucciones de objetos. Antes de seguir, se presentan sus listados. Si quiere, puede omitir su lectura, pues en el ejemplo se presentarán algunas operaciones con conjuntos y aplicaciones que podrían contener cualquier otra *cosa*.

cosa.h

```
1  /* Una clase para representar cualquier «cosa»,
2   * para llevar la traza de
3   * las actividades de objetos, en ejemplos.
4   */
5  #ifndef COSA_H_
6  #define COSA_H_
7
8  #include <iostream>
9  using std::ostream;
10
11 class Cosa {
12 public:
13   Cosa();
14   Cosa(const Cosa& o);
15   Cosa& operator =(const Cosa& o);
16   friend bool operator <(const Cosa& vi, const Cosa& vd)
17     { return vi.id < vd.id; }
18   friend bool operator ==(const Cosa& vi, const Cosa& vd)
19     { return vi.id == vd.id; }
20   ~Cosa();
21   friend ostream& operator <<(ostream& s, const Cosa& r);
22   friend class InformeCosa;
23 private:
24   static long int construcciones, asignaciones,
25     copias, destrucciones;
26   long int id;
27 };
28
29 struct GeneraCosa {
30   Cosa operator ()() { return Cosa(); }
31 };
32
33 /* Un «simpletrón». Sólo puede existir un objeto de esta
34  * clase, no más. Automáticamente dará informes cuando el
35  * programa termine.
36  */
```

```
37  class InformeCosa {
38  public:
39    ~InformeCosa();
40  private:
41    static InformeCosa ic;
42    InformeCosa() {} // constructor privado
43  };
44
45  #endif /* COSA_H_ */
```

Observe la clase *InformeCosa*. Al tener un único constructor y privado, no pueden crearse objetos de ella. Salvo porque su único atributo es... ¡del mismo tipo, de la misma clase! Esto sólo es posible, sin llegar a una definición recursiva, porque además es estático. Recuérdese que un atributo estático es en realidad como una variable global con ámbito de clase. Como los atributos estáticos han de definirse fuera de la clase, esto nos proporciona la posibilidad de poder crear un único objeto, y sólo uno, de ella. Además este atributo se definirá en el fichero de implementación, con lo que será global al programa. Cuando éste acabe, el destructor entrará en acción y mostrará lo que nos interesa. Como dice en el comentario del fichero de cabecera, una clase de este tipo se llama «simpletrón». He aquí la implementación:

cosa.cpp

```
1   /*
2    * Una clase para representar cualquier "cosa",
3    * para llevar la traza de
4    * las actividades de objetos, en ejemplos.
5    * Implementación.
6    */
7
8   #include "cosa.h"
9   #include <iostream>
10  using std::cout;
11  using std::ostream;
12  using std::endl;
13
14  Cosa::Cosa(): id(construcciones++)
15  {
16    cout << "d[" << id << "]";
17  }
18
19  Cosa::Cosa(const Cosa& o): id(o.id)
20  {
21    cout << "c[" << id << "]";
22    copias++;
23  }
24
```

```
25  Cosa& Cosa::operator =(const Cosa& o)
26  {
27    cout << "(" << id << ") = [" << o.id << "]";
28    id = o.id;
29    asignaciones++;
30    return *this;
31  }
32
33  Cosa::~Cosa()
34  {
35    cout << "~[" << id << "]";
36    destrucciones++;
37  }
38
39  ostream& operator <<(ostream& s, const Cosa& r)
40  {
41    return s << r.id;
42  }
43
44  // El destructor del «simpletrón».
45  InformeCosa::~InformeCosa()
46  {
47    cout << "\n---------------------\n"
48         << "Creaciones de cosas: " << Cosa::construcciones
49         << "\nConstrucciones de copia: " << Cosa::copias
50         << "\nAsignaciones: "      << Cosa::asignaciones
51         << "\nDestrucciones: "     << Cosa::destrucciones
52         << endl;
53  }
54
55  // Las inicializaciones de los atributos estáticos.
56  long int
57    Cosa::construcciones = 0L,
58    Cosa::asignaciones   = 0L,
59    Cosa::copias         = 0L,
60    Cosa::destrucciones  = 0L;
61  InformeCosa InformeCosa::ic;
```

La clase *Cosa* sólo tiene contadores estáticos, que llevan la cuenta de cuántas llamadas se han hecho al constructor predeterminado, al de copia, al destructor, y cuántas asignaciones, y un número (*id*) identificativo de cada objeto de la clase.

Ahora ya estamos en condiciones de volver al programa de ejemplo de aplicaciones y conjuntos.

basicasoc.cpp

```
/*
 * Operaciones básicas con conjuntos y aplicaciones
 */
#include "cosa.h"
#include <iostream>
#include <iterator>
#include <set>
#include <map>
using namespace std;

int main()
{
  Cosa vc[7];                    // vc: Vector de Cosa
  // cc: Conjunto de Cosa; añade elementos mediante el ctor.
  set<Cosa> cc(vc, vc + sizeof vc / sizeof(Cosa));
  // inserción ordinaria
  Cosa c;
  cc.insert(c);
  cout << endl;
  // comprobación de pertenencia
  cout << "cc.count(c) = " << cc.count(c) << endl;
  if (cc.find(c) != cc.end())
    cout << "c(" << c << ") pertenece a cc" << endl;
  // muestra elementos
  copy(cc.begin(), cc.end(), ostream_iterator<Cosa>(cout, " "));
  cout << "\n\n--------------" << endl;

  map<int, Cosa> ac; // ac: Aplicación de los enteros en Cosa
  for (int i = 0; i < 10; i++)
    ac[i];       // ¡construye los pares automáticamente!
  cout << "\n--------------" << endl;
  ac[10] = c;
  cout << "\n--------------" << endl;
  ac.insert(make_pair(47, c));
  cout << "\n--------------" << endl
       << "\n ac.count(10) = "
       << ac.count(10) << endl
       << "\n ac.count(11) = "
       << ac.count(11) << endl;
  map<int, Cosa>::iterator it = ac.find(6);
  if (it != ac.end())
    cout << "valor: " << it->second
         << " encontrado en «ac» en la posición 6" << endl;
  for (it = ac.begin(); it != ac.end(); it++)
    cout << it->first << ":" << it->second << ", "
         << "\n--------------" << endl;
}
```

El objeto *cc* de tipo *set<Cosa>* se ha creado mediante dos iteradores de un vector de bajo nivel de objetos *Cosa*, pero *set* también tiene un constructor predeterminado y uno de copia, y se puede pasar como parámetro un objeto que proporcione otra forma de efectuar comparaciones.

Tanto *set* como *map* tienen un método *insert()* para lo que su nombre indica: insertarles objetos, y hay un par de métodos para comprobar si un objeto está en un contenedor asociativo: *count()* y *find()*.

Dada una clave, *count()* dice cuántas veces está; evidentemente sólo 0 ó 1 en un *map* o *set*, pero puede estar más de 1 en un *multimap* o *multiset*.

En cambio, *find()* devolverá un iterador indicando la primera (y única, en el caso de *set* y *map*) ocurrencia de la clave que se le suministre, o el iterador que apunta a un elemento más allá del final en caso de no encontrar nada.

El *operator* `[]` se comporta aquí de una manera un tanto sorprendente. Observe el primer `for`. En primer lugar parece que se accede a unos elementos del contenedor a través de un índice, para no hacerse nada con ellos; si nos fijamos mejor vemos que esos elementos no existen siquiera. En efecto, cuando se aplica *operator* `[]` a un *map*, si el índice (clave) no existe, **se crea** el par correspondiente. En este caso se han creado 10 elementos cuyas claves son los enteros de 0 a 9 y cuyos valores son objetos *Cosa*, creados con su constructor predeterminado. Todo esto significa que si lo único que queremos es ver si existe algo sin crear un par nuevo, deberemos emplear *count()* o *find()* en lugar de *operator* `[]`.

Tampoco es buena idea mostrar todos los elementos de un *map* mediante un bucle `for` y el *operator* `[]`. Primero, porque las claves deberían ser enteros, como es el caso en este ejemplo, y segundo y principalmente, porque las claves pueden muy bien no ser secuenciales, con lo que se acabarían creando pares nuevos. Finalmente, como se puede comprobar estudiando la salida del programa una vez ejecutado, el *operator* `[]` no es muy eficiente, hay mucha «actividad» de construcción y destrucción de objetos.

En resumen: una aplicación (*map*) posee el interesante comportamiento de «crear un nuevo elemento si no existe ya», pero a costa de muchas creaciones y destrucciones extra, cuando se emplea *operator* `[]`.

Afortunadamente, hay alternativas, como se ve en el programa. El método *insert()* es más eficiente que *operator* `[]`. En un conjunto (*set*) sólo se inserta un objeto, pero en una aplicación se inserta un par clave-valor, por lo que *insert()* requiere como parámetro un *pair*. La función genérica *make_pair()* sirve a este respecto para crear uno, como indica su nombre.

Como se ha dicho antes, para buscar objetos en un *map*, puede emplearse *count()* para ver si una clave dada está en él, o *find()*, que devuelve un ite-

rador «apuntando» directamente al par encontrado. Como un *map* contiene elementos de tipo *pair* (es decir: como una aplicación contiene pares), esto es lo que produce el iterador cuando se desreferencia, de forma que nos interesa acceder a la clave y al valor por separado. Esto se consigue mediante los atributos de un *pair* llamados respectivamente *first* y *second*. Observe que `it->first` es equivalente a `(*it).first`, pues *operator* `->` está convenientemente sobrecargado para estos iteradores.

A continuación, estudiaremos más en detalle cada tipo de contenedor asociativo.

7.5.1. Aplicaciones: *map*

Un vector normal emplea un valor entero positivo como índice en un conjunto secuencial de elementos de algún tipo. Una aplicación puede considerarse un *vector asociativo* (también se suele llamar *diccionario*), donde se asocia un objeto con otro formándose una estructura de tipo vector (esto no quiere decir que esté implementada como un vector) de forma que en vez de seleccionar un elemento mediante un índice numérico, se emplea como índice un objeto. El ejemplo que sigue es un programa que cuenta las «palabras» en un fichero de texto, de forma que el índice es el objeto *string* que representa la palabra, y el valor correspondiente es el objeto que lleva la cuenta.

Así pues, a diferencia de otros contenedores que sólo contienen objetos individuales en sus elementos, en cada uno de un *map* hay dos cosas: la *clave*, que es lo que se busca y por lo que se indexa, y el *valor* que resulta de la búsqueda indexada por la clave.

Si se quiere recorrer el *map* y listar cada par clave-valor, se emplea un iterador que, cuando se desreferencia, produce un objeto de tipo *pair* que contiene la clave y el valor. Se accede a éstos mediante los atributos de *pair* llamados respectivamente *first* y *second*; es decir, *primero* y *segundo*.

Para insertar un elemento en un *map* hay que insertar pues evidentemente un par, es decir un objeto de tipo *pair*, pero éste se crea automáticamente como parte del *map* y su tipo se llama *value_type*, conteniendo la clave y el valor. De forma que una manera de insertar un nuevo elemento es crear un objeto de tipo *value_type*, cargarlo con los objetos apropiados y llamar al método *insert()* de *map*. La función auxiliar *make_pair()* es muy útil en estos casos. Otra forma es emplear el *operator* `[]` como se vio en el programa de ejemplo anterior: si el elemento buscado mediante la clave pasada a *operator* `[]` no existe, se crea y se inserta empleándose el constructor predeterminado del objeto valor. Con todo esto, veamos ya el programa que cuenta palabras.

cuentapalabras.cpp

```
/*
 * Cuenta el Nº de palabras en un texto empleando
 * una aplicación (map).
 */

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

typedef map<string, int>::const_iterator Iterador;

int main()
{
  map<string, int> cuenta;
  string palabra;
  while (cin >> palabra)
    ++cuenta[palabra];
  for (Iterador i = cuenta.begin(); i != cuenta.end(); i++)
    cout << i->first << ": " << i->second << endl;
}

```

Para abreviar, el programa lee de la entrada estándar (que podrá redireccionarse a un fichero de texto) y considera una «palabra» como cualquier conjunto de caracteres separados por blancos. Por supuesto se podría mejorar el programa haciendo que las palabras fueran realmente tales, y que leyera de un fichero que se suministrara como parámetro; pero estamos interesados solamente aquí en las aplicaciones o *map*s.

La línea crucial es

```
++cuenta[palabra];
```

donde se busca la *palabra* y, si se encuentra, se incrementa el objeto asociado con esa palabra, el valor, que es un entero *int* que lleva el número de ocurrencias de la palabra. Si no se encuentra, como ocurre para cada «palabra» nueva, entonces automáticamente se inserta una clave para la «palabra» y su valor asociado, que se crea con el constructor predeterminado. Un *int* es un tipo fundamental y no tiene constructor, pero ya se ha dicho que en C++ los tipos fundamentales se comportan a veces como si fueran clases definidas por el usuario, como si tuvieran constructores predeterminados y de copia. En este caso, el *int* se comporta como si tuviera constructor predeterminado, que lo inicializa al valor 0; a continuación se incrementa y toma el valor 1.

Para mostrar el *map*, lo recorremos con un iterador; al desreferenciarse, produce un objeto de tipo *pair* del cual seleccionamos la clave con *first* y el valor asociado con *second*. Podríamos mostrar el número de veces que está la palabra «la» con el *operator* `[]`, así:

```
cout << "la: " << cuenta["la"] << endl;
```

Se puede ver que una de las grandes ventajas del *map* es la claridad de la sintaxis; se está buscando el valor asociado a la palabra "la", no el valor correspondiente a un índice relativo a la posición desde el principio de un vector. Observe, sin embargo, que si no existe anteriormente la entrada correspondiente a "la", se crearía.

Por último se observa al ejecutar el programa que los elementos de un *map* están ordenados. Normalmente un *map* se implementa con alguna forma de árbol[8].

7.5.2. Aplicaciones con claves repetidas: *multimap*

Un *multimap* es un *map* que puede contener claves repetidas. A primera vista esto puede resultar raro, pero ocurre muy a menudo sin embargo. Por ejemplo, una libreta de direcciones y teléfonos: hoy día mucha gente tiene un teléfono fijo y uno móvil, aparte del teléfono o teléfonos del trabajo; por tanto puede haber varias entradas en la lista para el mismo nombre.

Un *multimap* no puede pues admitir indexado por claves como un *map*. Para acceder a los distintos valores con la misma clave se emplean las operaciones *equal_ range()*, *lower_ bound()* y *upper_ bound()*. Por ejemplo, para imprimir todos los números de teléfono de una persona, se haría algo así:

```
typedef multimap<string, unsigned long int> Libreta;

void mostrar_numeros(string persona, const Libreta& libreta)
{
  typedef Libreta::const_iterator I;
  pair<I, I> b = libreta.equal_range(persona);
  for (I i = b.first; i != b.second; i++)
    cout << i->second << endl;
}
```

Veamos un ejemplo completo. Supondremos que queremos estudiar la vida salvaje y saber dónde y cuándo se ha visto a un animal determinado. De

[8] Son muy empleados los árboles rojinegros.

forma que podemos ver muchos animales de la misma especie en sitios y momentos diferentes: el tipo de animal es la clave, se necesita un *multimap*.

vidasalvaje.cpp

```
/*
 * Estudio de la vida salvaje - simulación
 * Ejemplo de multimap
 */
#include <vector>
#include <map>
#include <string>
#include <algorithm>
#include <sstream>
#include <ctime>
#include <cstdlib>
#include <iterator>
#include <iostream>
using namespace std;

class PuntoDatos {
public:
  PuntoDatos(): x(0), y(0), t(0) {}
  PuntoDatos(int x, int y, time_t t): x(x), y(y), t(t) {}
  // Ctor. de copia, dtor. y operator = no hacen falta
  int abscisa() const { return x; }
  int ordenada() const { return y; }
  const time_t* instante() const { return &t; }
private:
  int x, y;  // coordenadas de localización
  time_t t;  // instante de la localización
};

string animal[] = { "urogallo", "castor", "marmota",
                    "comadreja", "ardilla", "perdiz nival",
                    "oso", "águila", "halcón", "cernícalo",
                    "ciervo", "nutria", "colibrí",
                  };
// tsa: Tamaño String Animal, el Nº de elementos del vector
const size_t tsa = sizeof animal / sizeof *animal;
vector<string> animales(animal, animal + tsa);

/* Toda la información está contenida en una
 * «Localizacion», que puede enviarse a un
 * flujo de salida
 */
typedef pair<string, PuntoDatos> Localizacion;

ostream& operator <<(ostream& os, const Localizacion& loc)
```

```
45 {
46   return os << loc.first << ": avistado en ("
47             << loc.second.abscisa() << ", "
48             << loc.second.ordenada() << "), instante = "
49             << ctime(loc.second.instante());
50 }
51
52 // Un generador de Localizacion'es
53 class GenLoc {
54 public:
55   GenLoc(vector<string>& an): animales(an) { srand(time(0)); }
56   Localizacion operator ()();
57 private:
58   vector<string>& animales;
59 };
60
61 Localizacion GenLoc::operator ()()
62 {
63   const int d = 100;
64   Localizacion resultado;
65   resultado.first  = animales[rand() % animales.size()];
66   resultado.second = PuntoDatos(rand() % d, rand() % d, time(0));
67   return resultado;
68 }
69
70 typedef multimap<string, PuntoDatos> Datos;
71 typedef Datos::iterator DIter;
72
73 int main()
74 {
75   Datos localizaciones;
76   generate_n(
77              inserter(localizaciones, localizaciones.begin()),
78              50,
79              GenLoc(animales)
80              );
81   // mostrarlo todo
82   copy(localizaciones.begin(), localizaciones.end(),
83        ostream_iterator<Localizacion>(cout)); // ctime() añade '\n'
84   // mostrar localizaciones de un animal seleccionado
85   while (true) {
86     cout << "Seleccione un animal, o 's' para salir: ";
87     for (vector<string>::size_type i = 0; i < animales.size(); i++)
88       cout << '[' << i << ']' << animales[i] << ' ';
89     cout << endl;
90     string respuesta;
91     getline(cin, respuesta);
92     if (respuesta== "s")
93       return 0;
```

```
94      istringstream r(respuesta);
95      int i;
96      r >> i;  // convierte a int
97      i %= animales.size(); // para evitar que esté fuera del rango
98      // los iteradores en «rango» denotan begin (principio),
99      // end (1 más allá del final) del rango concordante:
100     pair<DIter, DIter> rango =
101       localizaciones.equal_range(animales[i]);
102     copy(rango.first, rango.second,
103          ostream_iterator<Localizacion>(cout));
104   }
105 }
```

Todos los datos de una localización se encapsulan en la clase *PuntoDatos*, que es tan simple que podemos conformarnos con el constructor de copia, el destructor y el operador de asignación proporcionados por el compilador automáticamente. Guarda el instante de la localización como un «tiempo de calendario», según se define en la biblioteca estándar de C y C++.

En el vector de bajo nivel de *string*s llamado *animal*, cada *string* se construye con el constructor de conversión `string(const char*)` en la inicialización del vector. Pero como es más fácil y conveniente siempre emplear un *vector* de alto nivel, se crea uno a partir de éste de bajo nivel mediante el constructor de rango, `vector(iterator, iterator)` Para un vector de bajo nivel, un iterador es simplemente un puntero: `animal` es la dirección del primer elemento, del principio del vector de bajo nivel, mientras que `animal + tsa` apunta a un elemento más allá del final. En C y C++ esto es legal, mientras no se desreferencie el puntero; esto equivale a los métodos *end()*, que producen un iterador que apunta a un lugar más allá del final de una secuencia.

Los pares clave-valor que componen las localizaciones son el nombre del animal, en un *string*, y un *PuntoDatos* que dice cuándo y dónde ha sido visto. A un par así le llamamos *Localizacion*. Sobrecargamos el operador de inserción sobre este tipo para poder iterar sobre un *map* o *multimap* de *Localizacion*es imprimiéndolas.

Como este programa es sólo un ejemplo, un pequeño simulador, creamos la clase *GenLoc* para generar localizaciones aleatorias. Tiene el necesario *operator* `()` de un objeto función (esto se verá más en detalle en este capítulo), pero también tiene un constructor para capturar y guardar una referencia a un *vector*<*string*>, que es donde se guardan los nombres de los animales.

Por fin llegamos a lo interesante: un *Datos* es un *multimap* de pares *string*-*PuntoDatos*; o sea, que almacena localizaciones. Las claves son *string*s con los nombres de los animales, y los valores son los sitios e instantes donde se han avistado. Para un animal dado, puede haber varios avistamientos, y de

ahí que su nombre deba ser una clave repetida; por tanto hay que emplear un *multimap*.

Ya en *main()*, se define un objeto de tipo *Datos* y se insertan 50 avistamientos. En un programa real, estos datos se leerían de un fichero o se introducirían interactivamente, pero en este ejemplo simularemos los avistamientos con coordenadas aleatorias. Para las inserciones se emplea el algoritmo *generate_n()*, que asigna el resultado de llamar 50 veces al objeto función *GenLoc* sin parámetros[9] a cada elemento en el rango determinado por el primer parámetro hasta el primer parámetro más 49. El primer parámetro es una llamada a la función genérica *inserter()*, que devuelve un *insert_iterator*<*Localizaciones*>; en resumen: la llamada a *generate_n()* inserta, en el *multimap localizaciones*, 50 pares de tipo *Localizacion* generados aleatoriamente por `GenLoc::operator ()`.

A continuación se muestra todo el *multimap*; para ello se llama al algoritmo *copy()*, que recibe un rango en forma de dos iteradores y lo copia en el tercer parámetro, que es un iterador de salida que efectúa salida formateada de objetos de tipo *Localizacion* en este caso en un flujo de salida, *cout* en el ejemplo. Para ello el *operator* `<<` debe estar sobrecargado convenientemente, como se hizo antes.

Ahora se pregunta al usuario de qué animal quiere ver los avistamientos. Se le presenta la lista de animales, cada nombre precedido de un número, para que escoja. Con este número se indexa el *vector* de *animales* y el *string* resultante con el nombre del animal se pasa como parámetro al método *equal_range()* del *multimap localizaciones*. Este método devuelve un par (*pair*) de iteradores: el primero (*first*) «apunta» al principio del conjunto de pares concordantes, y el segundo (*second*) a un elemento más allá del final de dicho conjunto. Con esto, el último *copy()* nos mostrará los avistamientos de un determinado animal.

7.5.3. Conjuntos: *set*

Un conjunto (*set*) es un contenedor que sólo aceptará un ejemplar de cada objeto que se inserte en él. También se ordenan los elementos, y quizás está implementado como un árbol binario equilibrado para que las búsquedas sean rápidas. Un *set* puede verse por tanto como un contenedor asociativo *ordenado*, puesto que los elementos quedan clasificados; *simple*, puesto que es como un *map* donde sólo hay claves, y *único*, puesto que no admite más de una clave, o sea, no hay elementos repetidos.

El siguiente ejemplo muestra la lista de «palabras» de un texto, sin repe-

[9] Esto es: se crea un objeto temporal de tipo *GenLoc* con su constructor, llamémosle *temp*, y *generate_n()* llama 50 veces a *temp()*, o dicho de otra forma, a *temp*.*operator* `()`.

ticiones, ordenadas lexicográficamente. Para abreviar, como antes, se lee de la entrada estándar y se considera una «palabra» simplemente un conjunto de caracteres delimitados por blancos. Se deja como ejercicio al lector perfeccionar este programa y los similares.

Ejemplo:

listapalabras.cpp

```
/* Ejemplo de 'set'.
 * Muestra en la salida estándar una lista ordenada
 * de «palabras» únicas leídas desde la entrada estándar.
 */
#include <string>
#include <set>
#include <iostream>
#include <iterator>
using namespace std;

int main()
{
  set<string> palabras;
  string palabra;
  while (cin >> palabra)
    palabras.insert(palabra);
  copy(palabras.begin(), palabras.end(),
       ostream_iterator<string>(cout, "\n"));
}
```

Se observa que al insertar palabras, si una ya está en el conjunto, no se inserta de nuevo.

El siguiente ejemplo no hace nada útil pero ejercita más los conjuntos y los algoritmos que hay para ellos:

Ejemplo:

ej-set.cpp

```
/* Ejercicios sobre conjuntos (set)
 */
#include <set>
#include <algorithm>
#include <cstring>
#include <iostream>
```

```
7  #include <iterator>
8  using namespace std;
9
10 /* Objeto-función que implementa el operador < sobre cadenas
11  * de caracteres de bajo nivel (const char*)
12  */
13 struct menor
14 {
15   bool operator()(const char* s1, const char* s2) const
16     {
17       return strcmp(s1, s2) < 0;
18     }
19 };
20
21 int main()
22 {
23   const size_t n = 6;
24   const char* a[n] = {"Jano", "Marte", "Europa",
25                       "Prometeo", "Apolo", "Plutón" };
26   const char* b[n] = {"Cibeles", "Plutón", "Baco",
27                       "Marte", "Venus", "Apolo" };
28
29   set<const char*, menor> A(a, a + n);
30   set<const char*, menor> B(b, b + n);
31   set<const char*, menor> C;
32
33   cout << "A = { ";
34   ostream_iterator<const char*> salida(cout, ", ");
35   copy(A.begin(), A.end(), salida);
36   cout << "\b\b }" << endl;
37   cout << "B = { ";
38   copy(B.begin(), B.end(), salida);
39   cout << "\b\b }" << endl;
40
41   menor m;
42   cout << "Unión: A U B = { ";
43   set_union(A.begin(), A.end(), B.begin(), B.end(), salida, m);
44   cout << "\b\b }" << endl;
45
46   cout << "Intersección: A ^ B = { ";
47   set_intersection(A.begin(), A.end(), B.begin(), B.end(), salida, m);
48   cout << "\b\b }" << endl;
49
50   set_difference(A.begin(), A.end(), B.begin(), B.end(),
51                  inserter(C, C.begin()), m);
52   cout << "Diferencia: C = A - B = { ";
53   copy(C.begin(), C.end(), salida);
54   cout << "\b\b }" << endl;
55 }
```

7.5.4. Multiconjuntos: *multiset*

Se acaban de ver los conjuntos, donde sólo se permitía la existencia de un objeto con un determinado valor. Un multiconjunto en cambio permite la coexistencia de más de un objeto con un mismo valor. ¿Cómo entonces se entiende la noción de conjunto, donde uno pregunta «¿pertenece *este* elemento al conjunto?» Si hay más de un *este*, ¿qué sentido tiene la pregunta?

En efecto, no parece tener mucho sentido que haya dos o más objetos con el mismo valor si esos objetos repetidos son *exactamente* iguales, aunque quizá sí nos interese saber el número de repeticiones. En un multiconjunto un objeto duplicado puede tener algo que lo hace diferente de los otros; seguramente información que no se utilizó durante la comparación. Es decir: para la operación de comparación, los dos objetos son equivalentes, pero realmente su estado interno es distinto.

Como ejemplo, veamos una nueva versión del programa de contar «palabras», donde precisamente lo que nos interesará es el número de repeticiones.

Ejemplo:

mset-cuentapalabras.cpp

```
1  // Cuenta ocurrencias de «palabras» empleando un multiset
2  #include <string>
3  #include <set>
4  #include <iostream>
5  #include <iterator>                // distance()
6  using namespace std;
7
8  int main()
9  {
10   multiset<string> msetpals;
11   string palabra;
12   while (cin >> palabra)
13     msetpals.insert(palabra);
14   typedef multiset<string>::iterator MsetIt;
15   MsetIt it = msetpals.begin();
16   while (it != msetpals.end()) {
17     pair<MsetIt, MsetIt> p = msetpals.equal_range(*it);
18     int num = distance(p.first, p.second);
19     cout << *it << ": " << num << endl;
20     it = p.second;               // a la siguiente palabra
21   }
22 }
```

Cada «palabra» se inserta en un *multiset<string>*. Se crea un iterador inicializándolo con el principio del *multiset*; cuando se desreferencie (`*it`) producirá cada «palabra». El método *equal_range()* produce un par de iteradores al comienzo y al final de la palabra que se ha seleccionado; es decir, si hemos seleccionado la palabra *conjunto*, por ejemplo, y estaba repetida en el texto de entrada 7 veces, el primer iterador del par apunta a la primera ocurrencia de la palabra *conjunto* y el segundo iterador apunta a la séptima ocurrencia de dicha palabra. Para calcular ese número de ocurrencias empleamos la función *distance()*, declarada en `<iterator>`, que devuelve el número de elementos en un rango.

Por último, el iterador se hace apuntar a un elemento más allá del rango (*second*); o sea, a la siguiente palabra.

7.5.5. Resumen de contenedores asociativos ordenados

Los contenedores asociativos ordenados son *set*, *map*, *multimap* y *multiset*. Contienen elementos accesibles a través de una clave, y soportan la recuperación eficiente de elementos (valores) a través de esa clave. Poseen una relación de orden, *Compare*, que es el objeto de comparación para el contenedor asociativo y por omisión es el objeto función *less<Key>*. En las tablas siguientes, *ASOC* representa una de las cuatro clases anteriores: un contenedor asociativo. Por supuesto, también poseen todas las características de los contenedores en general.

Tipo	Descripción
key_type	el tipo de la clave (*Key*)
key_compare	el del objeto de comparación (*Compare*) para claves
value_compare	el del objeto de comparación para valores

Cuadro 7.6: Tipos en contenedores asociativos

7.6. Manejo de bits

Puede parecer sorprendente que un lenguaje como C, diseñado pensando sobre todo en programación de sistemas, a bajo nivel, que puede sustituir con ventaja muchas veces al ensamblador, famoso por su rico conjunto de operadores, especialmente de bits, no posea una representación nativa de números binarios. Se acerca con las representaciones en octal, hoy día poco empleadas, y en hexadecimal, que son meras facilidades para traducir entre decimal y binario, pero no se puede escribir una constante como por ejemplo

Constructor	Descripción
ASOC()	predeterminado, usa *Compare*
ASOC(*cmp*)	emplea *cmp* como el objeto de comparación
ASOC(*b_it*, *e_it*)	emplea el rango [*b_it*, *e_it*) usando *Compare*
ASOC(*b_it*, *e_it*, *cmp*)	emplea el rango [*b_it*, *e_it*) usando *cmp* como objeto de comparación

Cuadro 7.7: Constructores de contenedores asociativos

Llamada	Descripción
c.insert(t)	inserta *t* si ningún elemento tiene la misma clave que *t*; devuelve *pair*<*iterator*, *bool*> siendo *bool* `true` si *t* no estaba presente
c.insert(w_it, t)	inserta *t* con *w_it* como posición de comienzo para la búsqueda; falla en *set*s y *map*s si la clave ya está presente; devuelve la posición de la inserción
c.insert(b_it, e_it)	inserta los elementos en el rango especificado por *b_it* y *e_it*
c.erase(k)	quita elementos cuya clave es *k*, devolviendo el Nº de elementos borrados
c.erase(w_it)	borra el elemento apuntado
c.erase(b_it, e_it)	borra los elementos del rango

Cuadro 7.8: Inserción y borrado

Llamada	Descripción
c.find(k)	devuelve un iterador al elemento con la clave k; si no existe, devuelve el iterador del final
c.count(k)	devuelve el Nº de elementos con k
c.lower_bound(k)	devuelve un iterador al primer elemento con un valor mayor o igual a k
c.upper_bound(k)	devuelve un iterador al primer elemento con un valor mayor que k
c.equal_range(k)	devuelve un par de iteradores a *lower_bound()* y *upper_bound()*

Cuadro 7.9: Otros métodos

`0b10001010`[10]. C++ sigue sin tener una representación nativa para constantes literales enteras en binario, pero al menos se dispone de dos clases en la biblioteca estándar que facilitan el trabajo cuando hay que manipular un grupo de valores *verdadero-falso*, o *encendido-apagado*, o, en definitiva, 1-0. Estas clases son *bitset*<*N*> y *vector*<*bool*>. Principalmente difieren en lo siguiente:

1. El *bitset* maneja un número determinado y fijo de bits. El usuario establece ese número mediante el parámetro entero de la plantilla, y tiene que ser una constante conocida en tiempo de compilación. En cambio, el *vector*<*bool*> puede, como el resto de *vector*es, expandirse automática y dinámicamente para admitir cualquier número de valores *bool*.

2. El *bitset* está diseñado para la eficiencia en la manipulación de bits, no como un contenedor «normal». Así, no posee iteradores, y sí muchas operaciones específicas de manejo de bits. En cambio el *vector*<*bool*>, como especialización de *vector*, posee todas las operaciones de éste; la especialización sólo intenta ser eficiente en espacio de almacenamiento para *bool*.

No existe una conversión trivial entre estas dos clases, lo que sugiere que han sido diseñadas para propósitos diferentes.

[10] Aunque existen programas-filtro que aceptan programas escritos con este tipo de constantes y las traducen a constantes válidas en C, generando un fichero que puede compilarse con el compilador. Esto es de todas formas un parche, y un incordio. También existen parches para ciertos compiladores, para que admitan tales constantes.

7.6.1. Conjunto de bits: *bitset<N>*

La plantilla de *bitset<N>* requiere un parámetro entero N que es el número de bits a representar. Así, *bitset<10>* y *bitset<12>* son tipos diferentes, y no se permiten comparaciones ni asignaciones entre ellos por tanto.

Las posiciones de los bits se numeran de derecha a izquierda, como en la mayoría de *palabras* de los ordenadores, de forma que el peso de `b[`i`]` es 2^i. De todo esto se deduce que un *bitset* puede verse como un número binario de N bits.

Ejemplo:

```
bitset<10> b = 0X3DDUL;
```

posición:	9	8	7	6	5	4	3	2	1	0
b:	1	1	1	1	0	1	1	1	0	1

Un *bitset* proporciona prácticamente todas las operaciones de bits deseables, en una forma muy eficiente. Sin embargo, el tamaño de un *bitset* es siempre un múltiplo del tamaño de un (*unsigned*) *long*; la única conversión desde un *bitset* a un valor entero es a un *unsigned long*.

Como de costumbre, *bitset* se presenta en el espacio de nombres *std*, en la cabecera `<bitset>`.

Construcción

El constructor predeterminado crea un *bitset* con todos los bits a 0; existe el constructor de copia, y el operador de asignación.

Además, hay un constructor de conversión desde *unsigned long*, y por último otro constructor explícito que crea un *bitset* a partir de un *string* compuesto de caracteres `'0'` y `'1'`. Cualquier otro carácter provoca el lanzamiento de la excepción *invalid_argument*. Este último constructor admite dos parámetros más que permiten escoger una subcadena: la posición y el número de caracteres.

Ejemplo:

```
bitset<16> b1;          // 0000 0000 0000 0000
bitset<16> b2(13uL);    // 0000 0000 0000 1101
bitset<16> b3 = 13UL;   // igual
```

```
bitset<16> b4("1101");  // igual (conversión const char* -> string)
bitset<16> b5 = "1101"; // ERROR: no hay conversión char* -> bitset
bitset<16> b6("10101011101110", 7, 4); // igual, 1101
bitset<16> b7("n0g00d");// Mal: lanza excepción invalid_argument
bitset<16> b8(b2);      // copia de b2
bitset<16> b9 = b2;     // igual
bitset<32> b0 = b2;     // ERROR, tipos distintos
b1 = b2;                // asignación
```

Conversiones

No se definen operadores de conversión, pero sí dos funciones que sirven como tales; éstas permiten convertir un *bitset* a un *unsigned long* y a un *string*, y son, respectivamente, *to_ulong()* y *to_string()*.

La primera lanza la excepción *overflow_error* si los bits no son representables en un *unsigned long*.

La segunda necesita una sintaxis bastante fea debido a que *string* es una plantilla (en realidad, *string* es un sinónimo de *basic_string<char>*) y se necesita especificación explícita de parámetro de función genérica.

Ejemplo:

Empleando las definiciones del ejemplo anterior:

```
unsigned long int u = b2.to_ulong();
string s = b2.template to_string<char>();
```

Entrada/salida

Los operadores de inserción y extracción de flujo están sobrecargados de forma que aceptan un *bitset*.

Ejemplo:

```
bitset<8> b;
cin >> b;
cout << b << endl;
```

Si la entrada es

```
␣␣␣1100abc
```

b contendrá 1100, es decir, 12UL; el siguiente carácter que se lea del flujo será *a*. El número se muestra en binario; la salida de ese trozo de código sería

```
1100
```

Operaciones de bits

Acceso a un bit Se puede acceder a un bit individual mediante *operator* `[]`, cuya complejidad es $O(1)$. Como en C++ no se puede referenciar un solo bit, este operador devuelve un objeto de tipo *reference*, definido en *bitset* para este propósito. Este tipo tiene definida la asignación, la asignación con conversión desde *bool*, el *operator* `~`, la conversión a *bool* y el método *flip()*; todo esto permite manipular una *reference* (devuelta por *operator* `[]`) como un bit. El *operator* `[]` lanza la excepción *out_of_range* si la posición está fuera de rango.

Ejemplo:

```
bitset<16> b = 0xBAUL;  // 0xba = 186 en decimal = 10111010
bool vf = b[2];  // vf = (b.operator [](2)).operator bool()
b[2] = true;     // (b.operator [](2)).operator =(true)
b[1] = b[3];     // asignación de 2 reference's
b[0].flip();     // invierte el bit 0
vf = ~b[2];      // devuelve el bit 2 invertido
```

En todas las tablas que siguen, *a* y *b* son *bitsets*; *p* es una posición o índice, *n* es un entero positivo y *v* es un *bool*.

Operadores de bits Se sobrecargan los operadores de bits con el significado habitual. Los de la tabla 7.10 devuelven una referencia al propio objeto (`*this`).

Los desplazamientos son lógicos, no aritméticos; es decir, las vacantes se rellenan con ceros.

Los de la tabla 7.11 devuelven una copia del propio objeto (`*this`), tras aplicarle la operación pedida.

Operación	Descripción
a `&=` b	Y
a `|=` b	O
a `^=` b	O exclusivo
a `<<=` b	desplazamiento lógico a la izquierda
a `>>=` b	desplazamiento lógico a la derecha

Cuadro 7.10: *bitset*: Operadores de bits con asignación

Operación	Descripción
`~`a	complemento: invierte los bits
a `<<` n	desplaza los bits de a n posiciones a la izquierda
a `>>` n	desplaza los bits de a n posiciones a la derecha

Cuadro 7.11: Otros operadores de bits (miembros de *bitset*)

Por supuesto, está el resto de operadores de bits, pero no como funciones miembro sino externas a la clase. Todas devuelven por valor un *bitset* con el resultado de la operación. Se presentan en la tabla 7.12.

Operación	Descripción
a `&` b	Y
a `|` b	O
a `^` b	O exclusivo

Cuadro 7.12: Otros operadores de bits (externos a la clase *bitset*)

También se proporcionan algunas funciones miembro para manipular los bits; se ven en la tabla 7.13.

Otras operaciones Por último, se dispone de algunas útiles operaciones, y los operadores de comparación. Vea la tabla 7.14.

7.6.2. Vector de booleanos: *vector<bool>*

El tipo *vector<bool>* es una especialización total de la plantilla *vector*; esto significa que la implementación es distinta de la de los demás vectores. Esta especialización es necesaria para mejorar el almacenamiento: aunque una

Llamada	Descripción
b.set()	pone todos los bits a 1
b.set(p, v)	`b[p] = v` (si se omite *v* se supone 1)
b.reset()	pone todos los bits a 0
b.reset(p)	`b[p] = 0`
b.flip()	cambia el valor de cada bit
b.flip(p)	cambia el valor del bit *p*-ésimo

Cuadro 7.13: Métodos de *bitset* para manipulación de bits

Operación	Descripción
b.count()	número de bits a 1
b.size()	número total de bits
a `==` *b*	`true` si son iguales
a `!=` *b*	`true` si son distintos
b.test(p)	`b[p] == 1`
b.any()	`true` si algún bit está a 1
b.none()	`true` si ningún bit está a 1

Cuadro 7.14: Otras operaciones de *bitset*

variable *bool* normal requiere al menos un byte, la implementación ideal sería que cada elemento *bool* ocupara tan solo un bit. Esto implica que, internamente, un iterador de *vector<bool>* no puede ser un *bool**, sino que debe ser definido especialmente.

Las funciones de manipulación de bits son mucho más reducidas que las de *bitset*; sólo se ha añadido a *vector* *flip()*, que invierte todos los bits; no hay *set()* ni *reset()*. Cuando se emplea *operator* `[]` se obtiene un objeto de tipo *reference*, definido especialmente puesto que no se puede referenciar un simple bit. Este tipo *reference* posee también una operación *flip()* que invierte el bit resultante.

Ejemplo:

La siguiente función lee un *vector<bool>* desde la entrada estándar empleando un bucle con índices, y muestra el mismo *vector<bool>* en la salida estándar mediante un iterador.

```
void f(vector<bool>& v)
{
  for (vector<bool>::size_type i = 0; i < v.size(); i++)
    cin >> v[i];
  for (vector<bool>::const_iterator i = v.begin();
       i != v.end(); i++)
    cout << *i << ' ';
  cout << endl;
}
```

7.7. Objetos función

Un *objeto función* o *functor* es aquél que posee una sobrecarga del operador de llamada a función *operator* `()`. La posibilidad de definir objetos función (que pueden ser recibidos como parámetros y ser devueltos por una función) permite pensar en C++ como en un *lenguaje de primer orden*. Se dice así que los objetos función son *objetos de primer orden*.

> Recuérdese que *operator* `()` sólo puede sobrecargarse como función miembro no estática.

En STL muchos algoritmos emplean funciones proporcionadas por el programador (a través de objetos función) como medio de parametrizar su comportamiento. Así, es normal que, por ejemplo, para los algoritmos de búsqueda

y ordenación existan versiones a las que se les puede suministrar la relación de equivalencia o de orden a emplear. Esto permite lograr flexibilidad.

Al igual que los iteradores permiten abstraer el empleo habitual de los punteros ordinarios, los objetos función representan una abstracción de los punteros a función. De hecho, los algoritmos genéricos están definidos en STL de manera que puedan recibir punteros a función en lugar de objetos función[11], aunque emplear estos últimos reporta algunas ventajas; también es posible transformar un puntero a función en un objeto función.

Del mismo modo que en STL existen iteradores predefinidos, también existe una serie de objetos función estándar, que se encuentran en `<functional>`. En ocasiones, el programador puede emplear estos objetos función predefinidos como elementos de construcción de los suyos propios. Los objetos función estándar también son genéricos, esto es, sus clases son paramétricas.

7.7.1. Clasificación

En STL, los objetos función se clasifican en las siguientes categorías según el número de parámetros que reciben:

Generadores Poseen una sobrecarga de *operator* `()` que no recibe parámetros.

Objetos función unarios Poseen una sobrecarga de *operator* `()` que recibe un parámetro.

Objetos función binarios Poseen una sobrecarga de *operator* `()` que recibe dos parámetros.

Es común que los generadores empleen su estado interno para producir valores de acuerdo con unas reglas prefijadas.

Los objetos función unarios y binarios que devuelven un valor lógico se denominan *predicados*.

Nótese que es posible definir objetos función ternarios, etc.; no obstante, STL únicamente necesita objetos función de estas categorías. Para permitir incluir fácilmente información de tipo sobre los parámetros y el resultado de los objetos función ordinarios, `<functional>` define las siguientes clases:

```
template <typename P, typename R>
struct unary_function
{
```

[11] Esto es así porque el tipo del objeto función es un parámetro de la plantilla.

```
  typedef P argument_type;
  typedef R result_type;
};

template <typename P1, typename P2, typename R>
struct binary_function
{
  typedef P1 first_argument_type;
  typedef P2 second_argument_type;
  typedef R result_type;
};
```

Los objetos función que heredan públicamente de alguna de estas clases poseen información de tipo explícita sobre sus parámetros y resultado, denominándose *adaptables*, pues es necesaria esta información para poder emplear sobre ellos *adaptadores*, como se verá más adelante. Todos los objetos función estándar son adaptables.

Ejemplo:

Resultan muy útiles, sobre todo en simulación, los objetos función que generan números pseudoaleatorios o, en general, objetos pseudoaleatorios de un cierto tipo.

A continuación se define una clase para generar números pseudoaleatorios uniformemente distribuidos en el intervalo discreto $[0, n)$. Ésta puede emplearse como base en la construcción de generadores de números pseudoaleatorios para diferentes distribuciones discretas y continuas.

gna/gna.h

```
1  #ifndef GNA_H_
2  #define GNA_H_
3
4  class Gna {
5  public:
6    Gna(int s = 1);
7    int operator ()(int n);
8  private:
9    int x; // semilla y último valor generado de la secuencia
10 };
11
12 // Construcción de un GNA a partir de una semilla.
13 //
14 // La semilla debe pertenecer a un cierto intervalo discreto y,
15 // en concreto, no debe ser 0; véase Gna::operator ().
```

```
16
17 inline Gna::Gna(int s): x(s) { }
18
19 #endif
```

Obsérvese cómo se declara una sobrecarga unaria del operador de llamada a función. Esto hace que los objetos puedan ser empleados como objetos función unarios que trabajan con el tipo *int*.

De hecho, la clase podría haberse declarado de la siguiente forma, con lo que poseería *argument_type* y *result_type*, ambos *int*, y sus objetos función serían adaptables.

```
#include <functional>
using std::unary_function;

class Gna: public unary_function<int, int> {
  // ...
};
```

A continuación se define el operador, que es donde se realiza todo el trabajo.

gna/gna.cpp

```
1  #include "gna.h"
2
3  // Generador de números (pseudo)aleatorios uniformemente
4  // distribuidos en el intervalo discreto [0, n).
5  //
6  // Algoritmo:
7  //
8  // Emplearemos un LCG (Linear-Congruence Generator)
9  // multiplicativo x(n) = a x(n-1) mod m, con a = 7^5 y
10 // m = 2^31 - 1.
11 //
12 // Éste genera valores en el intervalo discreto [1, m), siempre
13 // que x(0) pertenezca a él (debido a esto el 0 no sirve como
14 // semilla). Los valores se normalizan al intervalo continuo
15 // [0, n) restando 1 y calculando el resto módulo n (ya que con
16 // este generador, los bits de menor orden son tan aleatorios
17 // como los de mayor orden).
18 //
19 // Es necesario que m quepa en un «int», por lo tanto
20 // supondremos que nuestros enteros son de, al menos, 32 bits.
21 //
22 // Referencias:
23 //
```

```
24  // Jain, Raj. "The Art of Computer Systems Performance Analysis.
25  // Techniques for Experimental Design, Measurement, Simulation,
26  // and Modeling". 1991.
27  //
28  // Knuth, Donald E. "The Art of Computer Programming. Vol. 2:
29  // Seminumerical Algorithms". 3ª ed. 1997.
30
31  int Gna::operator ()(int n)
32  {
33    // Constantes privadas del LCG
34    static const int a = 16807,      // 7^5
35                     m = 2147483647, // 2^31 - 1
36                     q = m / a,
37                     r = m % a;
38    // Cálculo por el método de Scrhage (impide el desbordamiento)
39    if ((x = a * (x % q) - r * (x / q)) < 0)
40      x += m;
41    // Normalización del resultado al intervalo discreto [0, n)
42    return (x - 1) % n;
43  }
```

El siguiente programa de prueba muestra cómo se emplea el objeto función.

gna/gna.cpp

```
1   #include <iostream>
2   #include <iomanip>
3   #include <vector>
4   #include "gna.h"
5   using namespace std;
6
7   #ifdef IOSTREAMS_NO_ESTANDAR
8   ostream& fixed(ostream& fs) { fs.setf(ios::fixed); return fs; }
9   #define width setw
10  #endif
11
12  int main()
13  {
14    // Genera m números en el intervalo [0, n)
15    Gna g;
16    const size_t m = 1000000, n = 6;
17    vector<int> cuenta(n);
18    for (size_t i = 0; i < m; ++i)
19      ++cuenta[g(n)];
20    // Muestra los datos obtenidos
21    cout << fixed << setprecision(3);
22    cout << "Número Frec. Abs. Frec. Rel.\n"
23         << "------ ---------- ----------\n";
```

```
24    for (size_t i = 0; i < n; ++i) {
25      int frecuencia = cuenta[i];
26      cout << width(6)  << i
27           << width(11) << frecuencia
28           << width(11) << frecuencia / static_cast<double>(m)
29           << endl;
30    }
31  }
```

7.7.2. Objetos función predefinidos

En `<functional>` se definen varias clases estándar de objetos función. Principalmente podemos distinguir las clases de predicados estándar (véanse las tablas 7.15 y 7.16) y las de objetos función aritméticos (véase la tabla 7.17). Todas estas clases son, por supuesto, paramétricas y sirven como envoltorios funcionales de los operadores estándar correspondientes.

Por ejemplo, *equal_to* se define en `<functional>` de manera similar a:

```
template <typename T>
struct equal_to: public binary_function<T, T, bool>
{
  bool operator ()(const T& x, const T& y) const { return x == y; }
};
```

Todas las demás clases de predicados binarios están definidas de manera análoga, sustituyendo == por el operador correspondiente. La clase *logical_not* sería algo así:

```
template <typename T>
struct logical_not: public unary_function<T, bool>
{
  bool operator ()(const T& x) const { return !x; }
};
```

Algo muy similar ocurre con las clases de objetos función aritméticos:

```
template <typename T>
struct plus: public binary_function<T, T, T>
{
  T operator ()(const T& x, const T& y) const { return x + y; }
};
```

```
// ...

template <typename T>
struct negate: public unary_function<T, T>
{
  T operator ()(const T& x) const { return -x; }
};
```

Nombre	Categoría	Descripción
equal_to	Binario	Igual
not_equal_to	Binario	Desigual
less	Binario	Menor estricto
less_equal	Binario	Menor o igual
greater	Binario	Mayor estricto
greater_equal	Binario	Mayor o igual

Cuadro 7.15: Clases de predicados de comparación estándar

Nombre	Categoría	Descripción
logical_and	Binario	Conjunción lógica
logical_or	Binario	Disyunción lógica
logical_not	Unario	Negación lógica

Cuadro 7.16: Clases de predicados lógicos estándar

Nombre	Categoría	Descripción
plus	Binario	Adición
minus	Binario	Substracción
multiplies	Binario	Producto
divides	Binario	Cociente
modulus	Binario	Resto
negate	Unario	Opuesto

Cuadro 7.17: Clases de objetos función aritméticos estándar

7.7.3. Adaptadores de objetos función

Los *adaptadores* son funciones que permiten obtener fácilmente nuevos objetos función a partir de otros preexistentes (e incluso de funciones ordinarias). Por consiguiente, un adaptador puede considerarse como un *objeto de segundo orden*, ya que recibe un objeto función y devuelve otro.

Adaptadores de punteros a función Permiten convertir un puntero a función en un objeto función.

Negadores Permiten invertir el significado lógico de un predicado.

Ligadores Permiten ligar un parámetro de un objeto función a un valor concreto.

> También existen *adaptadores de funciones miembro*, que permiten construir objetos función a partir de funciones miembro, pero no los trataremos aquí.

Por cada uno de ellos se proporciona una *función de construcción* que crea objetos del tipo apropiado; los nombres que se emplean y su descripción pueden verse en la tabla 7.18.

Estas funciones de construcción reciben como primer parámetro el puntero a función (en el caso de los adaptadores de punteros a función) o el objeto función apropiado (en el caso de negadores y ligadores). Las correspondientes a los ligadores reciben además un segundo parámetro: el valor a ligar.

Para que un objeto función pueda ser ligado o negado se necesita que sea «adaptable», es decir, que posea la información de tipo que proporciona la clase *unary_function* o *binary_function*, según el objeto función sea unario o binario.

Para lograr este efecto cuando se define un nuevo objeto función, la forma más sencilla es heredar públicamente de la clase correspondiente (tal y como lo hacen los objetos función estándar).

Ejemplo:

El programa que se presenta a continuación muestra cómo crear un predicado binario para comprobar la paridad de un número. Para ello se define una clase que que posee una sobrecarga del operador de llamada a función.

par/par.cpp

Tipo	Función	Descripción
pointer_to_unary_function	*ptr_fun()*	Adaptador de puntero a función unaria
pointer_to_binary_function	*ptr_fun()*	Adaptador de puntero a función binaria
unary_negate	*not1()*	Negador de predicado unario
binary_negate	*not2()*	Negador de predicado binario
binder1st	*bind1st()*	Ligador del 1er parámetro
binder2nd	*bind2nd()*	Ligador del 2º parámetro

Cuadro 7.18: Adaptadores, negadores y ligadores estándar

```
1  #include <iostream>
2  #include <functional>
3
4  int main()
5  {
6    using namespace std;
7    unary_negate<binder2nd<modulus<int> > >
8      par = not1(bind2nd(modulus<int>(), 2));
9
10   cout << "n = ";
11   int n;
12   cin >> n;
13   cout << n << " es " << (par(n) ? "par" : "impar") << endl;
14 }
```

El predicado unario *par* permite comprobar si un número entero es par, ya que *modulus<int>()* es un objeto función que calcula el resto de la división de dos enteros, *bind2nd()* liga su segundo parámetro al valor 2, con lo que el objeto función resultante calcula el resto módulo 2 (0 si el número es par y 1 si es impar) y *not1()* lo interpreta como un predicado invirtiendo su significado (*true* si el número es par y *false* si es impar).

Obsérvese lo complejo del tipo de *par* en su definición. Es el precio que hay que pagar para que todo este mecanismo funcione en tiempo de compilación y sea tremendamente eficiente.

Podríamos haber logrado este mismo efecto creándolo directamente:

```
struct Par: public unary_function<int, bool>
{
  bool operator ()(int x) const { return x % 2 == 0; }
};

// ...

Par par;
```

7.8. Algoritmos

C++ permite la posibilidad de emplear el paradigma de la programación genérica por medio de los *algoritmos genéricos*. Estos algoritmos son «genéricos» en distintas vertientes:

1. Son paramétricos.
2. Reciben iteradores, no contenedores específicos.
3. Pueden, si es necesario, recibir objetos función.

Con los iteradores se consigue desacoplar a los algoritmos de las estructuras de datos que manipulan; con los objetos función, se consigue abstraer su funcionamiento.

Al realizarse estas dos abstracciones por medio del polimorfismo en tiempo de compilación (parametrización), se obtiene una eficiencia sin parangón en otros lenguajes de programación genérica.

Los algoritmos genéricos estándar que aparecen en STL, se encuentran en `<algorithm>`. Se describirá a continuación, brevemente, la mayoría de ellos.

7.8.1. Algoritmos observadores elementales

Los algoritmos observadores operan sobre un rango de iteradores sin modificar sus elementos, permitiendo obtener información sobre ellos.

Todos, salvo indicación en contra, reciben el rango de iteradores sobre el que trabajan como primer y segundo parámetros. Denotaremos por n a su número de elementos.

Aplicación de objeto función

for_ each()

> Recorre secuencialmente el rango aplicando el objeto función que recibe como último parámetro a cada elemento. El resultado de la aplicación, si lo hay, se desecha. Devuelve el objeto función.

Esta operación realiza exactamente n aplicaciones del objeto función.

Ejemplo:

Supongamos que *Figura* es un tipo polimórfico con una operación *mostrar()*, el siguiente fragmento permitiría mostrar todas las figuras almacenadas en una lista.

```
list<Figura*> figuras;
// ...
for_each(figuras.begin(), figuras.end(), mem_fun(Figura::mostrar));
```

La función de construcción *mem_fun()* adapta la función miembro *mostrar()*, construyendo el objeto función apropiado.

Búsqueda secuencial

Los algoritmos de búsqueda secuencial recorren un rango de iteradores buscando el primer elemento que cumple una determinada propiedad y devolviendo un iterador a él. En caso de no existir un elemento tal, devuelven el iterador que indica el final del rango.

find()/find_ if()

> Buscan el valor que reciben como tercer parámetro. La primera versión emplea *operator* == para la comparación y la segunda un predicado suministrado como último parámetro.

find_ first_ of()

> Busca cualquiera de los valores del rango que recibe como tercer y cuarto parámetros. Posee dos versiones: la primera emplea *operator* == para la comparación y la segunda un predicado suministrado como último parámetro.

adjacent_find()

Busca dos valores adyacentes repetidos. Posee dos versiones: la primera emplea *operator* == para la comparación y la segunda un predicado suministrado como último parámetro.

La complejidad, en número de comparaciones realizadas, es lineal en el peor caso para todos estos algoritmos; *find_first_of()* realiza a lo sumo $m \cdot n$ comparaciones, siendo m el número de elementos del segundo rango.

Cuenta

count()/*count_if()*

Cuentan el número de veces que aparece el valor que reciben como tercer parámetro. La primera versión emplea *operator* == para la comparación y la segunda un predicado suministrado como último parámetro.

Realizan exactamente n comparaciones.

Comparación

Estos algoritmos comparan dos rangos y poseen dos versiones: la primera emplea para la comparación el operador correspondiente y la segunda un predicado suministrado como último parámetro.

Salvo en el caso de *lexicographical_compare()*, del segundo rango únicamente se especifica el primer iterador. STL emplea este convenio con los algoritmos que requieren dos rangos de igual longitud; bastan tres iteradores para describir ambos rangos.

equal()

Devuelve *true* si los dos rangos son iguales elemento a elemento y *false* en caso contrario.

mismatch()

Si los dos rangos difieren, devuelve un par (*pair*) de iteradores con la primera posición de cada rango en la que lo hacen. Si no difieren, devuelve como primer elemento del par un iterador al final del primer rango.

lexicographical_compare()

Devuelve *true* si el primer rango es lexicográficamente[12] menor que el

[12] En los diccionarios se emplea un orden lexicográfico sobre las palabras.

segundo y *false* en caso contrario.

El número de comparaciones en el peor caso es n para *equal()* y *mismatch()*, y $2 \cdot \text{mín}\{m, n\}$ para *lexicographical_compare()*[13], siendo m el número de elementos del segundo rango.

Mínimos y máximos

Las funciones que se describen a continuación poseen dos versiones: la primera emplea para la comparación *operator* < y la segunda un predicado suministrado como último parámetro.

min()

Recibe dos elementos, no un rango; devuelve el primero, salvo que el segundo sea menor, en cuyo caso se devuelve este último.

max()

Recibe dos elementos, no un rango; devuelve el primero, salvo que sea menor que el segundo, en cuyo caso se devuelve este último.

min_element()

Recibe un rango de iteradores monodireccionales y devuelve un iterador al primer elemento del rango tal que no exista otro menor que él.

max_element()

Recibe un rango de iteradores monodireccionales y devuelve un iterador al primer elemento del rango que no sea menor a ningún otro.

Las funciones *min_element()* y *max_element()* realizan exactamente $n - 1$ comparaciones, salvo que el rango esté vacío, en cuyo caso no hacen ninguna y devuelven un iterador al final del rango.

Ejemplo:

La siguiente función muestra cómo se puede programar el algoritmo de ordenación por selección utilizando iteradores, la función *min_element()* que se acaba de comentar y la función *swap()* que se comenta en §7.8.2 y que recibe dos elementos, no un rango, y los intercambia.

[13] Esto es así porque la equivalencia se comprueba empleando *dos* comparaciones menor.

seleccion/ordenacion.h

```
#include <algorithm>

template <typename Iterador>
void ordenacion_por_seleccion(Iterador principio, Iterador fin)
{
  using namespace std;
  for (Iterador i = principio; i != fin; ++i) {
    Iterador j = min_element(i, fin);
    swap(*i, *j);
  }
}
```

Obsérvese el empleo de *min_element()* y cómo, claramente, el algoritmo resultante sólo requiere iteradores monodireccionales y realiza $\Theta(n^2)$ comparaciones.

7.8.2. Algoritmos modificadores elementales

Los algoritmos modificadores trabajan principalmente con rangos de iteradores cuyos elementos modifican. Algunos poseen versiones terminadas con el sufijo *_copy* o *_copy_if*; éstas reciben un iterador de salida adicional y permiten mantener inalterado el rango original, obteniéndose el resultado sobre el rango de salida que determinan dicho iterador y la longitud del rango de entrada.

Evidentemente, el rango destino de la copia debe ser válido y corresponder a una zona de memoria convenientemente inicializada y de suficiente espacio. Si se desea realizar inserción, en lugar de copia, puede combinarse su empleo con el de un insertor.

Copia

Estos algoritmos permiten realizar la copia de un rango a otro. Mientras *copy()* recibe dos iteradores de entrada y uno de salida, *copy_backwards()* recibe tres iteradores bidireccionales.

copy()

> Asigna secuencialmente cada elemento del rango de entrada a su correspondiente en el rango de salida.

copy_ backwards()

Análoga a *copy()*; realiza la copia en orden inverso. A diferencia de ésta, su tercer iterador marca el final del segundo rango, no el principio.

Este es el único algoritmo de la STL donde un rango se determina a partir del iterador que marca su fin.

La complejidad de ambos algoritmos, medida en número de asignaciones, es lineal; se realizan exactamente n asignaciones.

Intercambio

swap()

Recibe dos elementos, no un rango, y los intercambia.

swap_ ranges()

Recibe tres iteradores monodireccionales que determinan dos rangos disjuntos e intercambia cada elemento del primero con su correspondiente en el segundo.

Reemplazo y transformación

replace()/replace_ if()/ replace_ copy()/replace_ copy_ if()

Reemplazan en un rango de iteradores monodireccionales todos los elementos equivalentes a uno dado, o que cumplen un cierto predicado, por otro.

transform()

Existen dos versiones. La primera permite aplicar un objeto función unario a los elementos de un rango de entrada y copiar los resultados en un rango de salida que puede coincidir exactamente con él, *pero no solaparse de otro modo*. La segunda, es una versión binaria que, en vez de dos, recibe tres iteradores de entrada (es decir, dos rangos de entrada) y un objeto función binario.

Ejemplo:

En el siguiente fragmento se suman dos listas de igual longitud de números en coma flotante de doble precisión dejando el resultado en la primera de ellas. A continuación, se duplica el valor de los elementos de ésta. Por último, se reemplazan por cero aquellos elementos negativos.

```
list<double> a, b;
// ...
transform(a.begin(), a.end(), b.begin(), a.begin(),
          plus<double>());
transform(a.begin(), a.end(), a.begin(),
          bind1st(multiplies<double>(), 2));
replace_if(a.begin(), a.end(),
           bind2nd(less<double>(), 0.0), 0.0);
```

Si en vez de reemplazar por ceros los negativos en la primera lista, se quiere preservar ésta y producir el resultado en la segunda, se emplearía *replace_ copy_ if()*:

```
replace_copy_if(a.begin(), a.end(), b.begin(),
                bind2st(less<double>(), 0.0), 0.0);
```

Relleno

fill()

Dado un un rango de iteradores monodireccionales y un valor, asigna el valor a cada elemento del rango.

fill_ n()

Dado un iterador de salida, un tamaño n y un valor, asigna el valor a los n primeros elementos a partir del iterador.

generate()

Recibe un rango de iteradores monodireccionales y un generador (un objeto función sin parámetros). Ejecuta el generador una vez por cada elemento del rango y asigna a cada uno el resultado correspondiente.

generate_ n()

Recibe un iterador de salida, un tamaño n y un generador. Ejecuta el generador n veces y asigna a cada elemento a partir del iterador el resultado correspondiente.

Presumiblemente, cada ejecución del generador producirá un valor posiblemente distinto y dependiente de su estado interno.

Ejemplo:

El siguiente fragmento muestra 15 números aleatorios en la salida estándar:

```
generate_n(ostream_iterator<int>(cout, " "), 15, rand);
cout << endl;
```

Si, en vez de esto, queremos rellenar una lista, podemos hacer:

```
list<int> numeros(15);
generate(numeros.begin(), numeros.end(), rand);
```

o incluso:

```
generate_n(numeros.begin(), numeros.size(), rand);
```

Eliminación

remove()/remove_if()/ remove_copy()/remove_copy_if()

> Reorganiza los elementos de un rango de iteradores monodireccionales y devuelve un nuevo iterador final de manera que todos los elementos equivalentes a uno dado, o que cumplen un cierto predicado, quedan excluidos del nuevo rango. Las versiones de copia no transfieren los elementos excluidos.

unique()/unique_copy()

> Reorganiza los elementos de un rango de iteradores monodireccionales y devuelve un nuevo iterador final de manera que *todos los elementos, salvo el primero, en cada secuencia consecutiva de elementos equivalentes*, quedan excluidos del nuevo rango. Las versiones de copia no transfieren los elementos excluidos.
>
> Cada una dispone de dos versiones, la primera emplea *operator* `==` para la comparación y la segunda un predicado suministrado como último parámetro.

Las funciones *unique()* y *unique_copy()* tienen su mayor utilidad en la eliminación de duplicados en rangos ordenados, donde todos los elementos equivalentes aparecen consecutivos.

Ejemplo:

El fragmento de código que sigue, ordena un vector y muestra sus elementos excluyendo las repeticiones.

```
vector<int> v;
// ...
sort(v.begin(), v.end());
unique_copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << endl;
```

Permutación

Los siguientes algoritmos trabajan intercambiando de algún u otro modo los elementos presentes en un rango, obteniendo por consiguiente una permutación de éstos.

Todos estos algoritmos trabajan con rangos de iteradores bidireccionales, excepto *rotate()* y *rotate_copy()* que sólo requieren que sean monodireccionales.

Los algoritmos *next_permutation()* y *prev_permutation()* poseen dos versiones: la primera emplea como orden débil estricto *operator* < y la segunda un predicado suministrado como último parámetro.

reverse()/*reverse_copy()*

> Invierte los elementos del rango, empleando exactamente $\lfloor \frac{n}{2} \rfloor$ intercambios de elementos.

rotate()/*rotate_copy()*

> Recibe tres iteradores que determinan dos rangos consecutivos. Rota el rango completo a la izquierda tantas veces como elementos tenga el primer rango. Desde otro punto de vista, el algoritmo «intercambia» ambos rangos, esto es, reorganiza los elementos de manera que los del segundo rango aparezcan delante de los del primero.
>
> La función *rotate()* emplea, a lo sumo, n intercambios; no obstante, el número exacto de asignaciones varía según se trabaje con iteradores monodireccionales, bidireccionales o de acceso directo (aunque siempre en $O(n)$). Por el contrario, *rotate_copy()* realiza exactamente n asignaciones.

next_permutation()

Genera la permutación que sucede en orden lexicográfico a la determinada por los elementos del rango. Si ésta era ya la última (el rango ordenado inversamente) genera la primera (el rango ordenado) y devuelve *false*; en caso contrario, devuelve *true*. Emplea, a lo sumo, $\lfloor \frac{n}{2} \rfloor$ intercambios.

prev_permutation()

Genera la permutación que precede en orden lexicográfico a la determinada por los elementos del rango. Si ésta era ya la primera (el rango ordenado) genera la última (el rango ordenado inversamente) y devuelve *false*; en caso contrario, devuelve *true*. Emplea, a lo sumo, $\lfloor \frac{n}{2} \rfloor$ intercambios.

random_shuffle()

Baraja aleatoriamente los elementos de un rango de iteradores de acceso directo. El resultado es que, tras el algoritmo, el rango representará una de las $n!$ posibles permutaciones de los elementos. Realiza exactamente $n - 1$ intercambios para rangos no vacíos.

Existen dos versiones: una utiliza un generador de números pseudoaleatorios interno y la otra un objeto función unario capaz de generar una distribución uniforme en el intervalo discreto $[0, p)$, siendo p el valor de su parámetro.

La probabilidad de que se genere cualquier permutación es $\frac{1}{n!}$, es decir, el resultado está uniformemente distribuido[14] en el conjunto de las permutaciones de los n elementos.

Los algoritmos *next_permutation()* y *prev_permutation()* realizan, a lo sumo, $\lfloor \frac{n}{2} \rfloor$ intercambios de elementos.

Ejemplo:

El siguiente programa genera todas las permutaciones de una palabra, que recibe a través de la línea de órdenes como primer parámetro.

El primer bucle se encarga de ir permutando la cadena original hasta que ésta queda ordenada. El segundo, sigue generando permutaciones mientras no se obtenga la cadena original.

[14]Idealmente. Para ello el generador de números pseudoaleatorios debe producir a su vez distribuciones realmente uniformes de valores y, lo que es más difícil, debe poseer un periodo extraordinariamente grande.

permutaciones/permutaciones.cpp

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

int main(int argc, char** argv)
{
  if (argc != 2) {
    cerr << "Modo de empleo: " << *argv << " palabra" << endl;
    return 1;
  }
  string c(*++argv);
  do {
    cout << c << endl;
  } while(next_permutation(c.begin(), c.end()));
  while (c != *argv) {
    cout << c << endl;
    next_permutation(c.begin(), c.end());
  }
}
```

Otra posibilidad es ordenar primero la cadena; así el segundo bucle sobraría.

Partición

Los algoritmos de partición reciben un rango de iteradores y un predicado unario. Su función es reorganizar los elementos del rango de manera que los que satisfagan el predicado precedan a los que no lo hagan. El predicado divide, pues, a los elementos del rango en dos grupos.

partition()

> Realiza la partición de un rango de iteradores bidireccionales atendiendo a un predicado unario. Realiza exactamente n aplicaciones del predicado y, a lo sumo, $\lfloor \frac{n}{2} \rfloor$ intercambios.

stable_partition()

> Realiza la partición de un rango de iteradores monodireccionales atendiendo a un predicado unario. La partición es estable, es decir, se preserva el orden relativo entre los elementos de cada grupo.

Es un algoritmo adaptativo: realiza exactamente n aplicaciones del predicado, pero el número de intercambios varía según la memoria disponible. Si existe suficiente memoria disponible, realiza $O(n)$ intercambios; en caso contrario, aumenta a $O(n \cdot \log n)$.

Ambos algoritmos devuelven el iterador que separa los dos grupos de elementos.

7.8.3. Algoritmos numéricos generalizados

Quizás el adjetivo «numéricos» no sea del todo apropiado. Estos algoritmos son tan generales que poseen una segunda versión que recibe adicionalmente los objetos función necesarios para abstraer las operaciones que intervienen.

accumulate()

Recibe un rango de entrada y un valor inicial. Calcula secuencialmente la suma del valor inicial con los elementos del rango. Formalmente, calcula $v + \sum_{i=0}^{n-1} a_i$, siendo v el valor inicial y los a_i los elementos del rango de entrada.

inner_product()

Recibe tres iteradores, que determinan dos rangos de entrada, y un valor inicial. Calcula secuencialmente la suma del valor inicial con el producto de los elementos de ambos rangos. Formalmente, calcula $v + \sum_{i=0}^{n-1} a_i \cdot b_i$, siendo v el valor inicial y los a_i, b_i los elementos de los rangos de entrada.

partial_sum()

Recibe un rango de entrada y un iterador de salida. Calcula las sumas parciales y asigna los valores a los sucesivos elementos del rango determinado por el iterador de salida. Formalmente, calcula $s_i = \sum_{j=0}^{n-1} a_j$, $0 \leq i < n$, siendo los a_j los elementos del rango de entrada y los s_i los del rango de salida.

adjacent_difference()

Recibe un rango de entrada y un iterador de salida. Calcula las diferencias finitas de primer orden (Δ^1) y asigna sus valores a los sucesivos elementos del rango determinado por el iterador de salida.

De otro modo, calcula:

$$s_i = \begin{cases} a_0, & i = 0 \\ \Delta^1 a_{i-1} = a_i - a_{i-1}, & 1 \leq i < n \end{cases}$$

siendo los a_i los elementos del rango de entrada y los s_i los del rango de salida.

Tal como se ha explicado, $+$, $\cdot$ y $-$ pueden ser, en general, operaciones cualesquiera. Esto hace que estos algoritmos sean extraordinariamente potentes, ya que es posible utilizarlos en contextos muy diversos.

Ejemplo:

El programa que aparece a continuación emplea *accumulate()* para mostrar en la salida estándar el resultado de concatenar sus parámetros de la línea de órdenes.

Nótese cómo se escoge como valor inicial el elemento neutro de la operación de concatenación de cadenas, esto es, la cadena vacía.

concatena/concatena.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <algo.h> // debería ser <algorithm>
4
5  int main(int argc, char** argv)
6  {
7    using namespace std;
8    cout << accumulate(argv + 1, argv + argc, string()) << endl;
9  }
```

Ejemplo:

El siguiente fragmento calcula el producto escalar de dos listas de números complejos:

```
complex_double r;
list<complex_double> l1, l2;
// ...
r = inner_product(l1.begin(), l1.end(), l2.begin(), 0.0);
```

La longitud de la segunda lista ha de ser, al menos, igual a la de la primera. Observe que `0.0` se convierte implícitamente a *complex_double*.

7.8.4. Ordenación y operaciones en rangos ordenados

Todos estos algoritmos poseen dos versiones: la primera emplea para la comparación *operator* < y la segunda un predicado suministrado como último parámetro.

Ordenación

sort()

Ordena un rango de iteradores de acceso directo en $O(n \cdot \log n)$ comparaciones en el promedio, aunque en el peor caso puede realizar $O(n^2)$ comparaciones.

Esto permite al implementador emplear alguna variante del algoritmo de Hoare (*quicksort*) para su implementación.

No obstante, el mismo año en que se aprobó el Estándar de C++, Musser diseñó un nuevo algoritmo, conocido como *ordenación introspectiva* (*introsort*), que, teniendo complejidad $O(n \cdot \log n)$ en el peor caso, es en la práctica tan rápido como el de Hoare en el promedio.

De hecho, algunas de las mejores implementaciones de la STL incluyen ya este algoritmo.

stable_sort()

Análogo a *sort()*, pero manteniendo el orden relativo de los elementos equivalentes. Es decir, la ordenación es estable.

Es un algoritmo adaptativo[15]: si existe suficiente memoria disponible su complejidad es $O(n \cdot \log n)$ comparaciones; en caso contrario, aumenta a $O(n \cdot \log^2 n)$.

nth_element()

Recibe tres iteradores de acceso directo que determinan dos rangos consecutivos y reorganiza los elementos de manera que en la posición del segundo iterador quede el elemento que la ocuparía si el rango completo estuviera ordenado. Además se garantiza que los elementos del primer rango no serán mayores a los del segundo.

Realiza un número de comparaciones lineal en el promedio.

[15] Normalmente se emplea una variante adaptativa del algoritmo de ordenación por fusión (*mergesort*).

Ejemplo:

La siguiente función muestra cómo se puede programar una variante del algoritmo de ordenación de Hoare que emplea como pivote la mediana.

rapida/ordenacion.h

```
#include <algorithm>
#include <functional>

template <typename Iterador>
void ordenacion_rapida(Iterador principio, Iterador fin)
{
  using namespace std;
  if (fin - principio > 1) {
    Iterador mitad = principio + (fin - principio) / 2;
    nth_element(principio, mitad, fin);
    ordenacion_rapida(principio, mitad);
    ordenacion_rapida(mitad, fin);
  }
}
```

Obviamente, el algoritmo resultante requiere iteradores de acceso directo. Nótese el empleo de *nth_ element()* para seleccionar la mediana y pivotar sobre ella.

Fusión

Los algoritmos de fusión permiten obtener un rango ordenado a partir de otros dos incluyendo todos sus elementos.

La fusión es estable. Esto significa que se preserva el orden relativo de los elementos de cada rango de entrada y que si un elemento tiene equivalentes en ambos, los del primero precederán a los del segundo.

El algoritmo *merge()* recibe dos rangos de iteradores de entrada (de m y n elementos) y un iterador de salida. Se requiere además que los rangos de entrada no se solapen con el de salida.

En el caso de *inplace_ merge()*, se reciben tres iteradores que determinan dos rangos consecutivos de iteradores bidireccionales (en total, n elementos). El rango resultante es el determinado por el primer iterador y el último.

merge()

Fusiona dos rangos ordenados en no más de $m + n - 1$ comparaciones

(ninguna si ambos están vacíos).

inplace_merge()

Fusiona dos rangos ordenados en $O(n \cdot \log n)$ comparaciones. Es un algoritmo adaptativo: su complejidad puede mejorar sustancialmente (no más de $n - 1$ comparaciones para rangos no vacíos) dependiendo del espacio de memoria que consiga reservar. El algoritmo es «in situ» (o *in place*) en el sentido de que *puede* operar con complejidad espacial constante; pero no lo hará si hay memoria suficiente.

Ejemplo:

La siguiente función muestra cómo programar el algoritmo de ordenación por fusión (*mergesort*).

fusion/ordenacion.h

```
1  #include <algorithm>
2
3  template <typename Iterador>
4  void ordenacion_por_fusion(Iterador principio, Iterador fin)
5  {
6    using namespace std;
7    if (fin - principio > 1) {
8      Iterador mitad = principio + (fin - principio) / 2;
9      ordenacion_por_fusion(principio, mitad);
10     ordenacion_por_fusion(mitad, fin);
11     inplace_merge(principio, mitad, fin);
12   }
13 }
```

Nótese cómo el algoritmo requiere iteradores de acceso directo. Gracias a esto, la complejidad de la descomposición (el cálculo de *mitad*) es constante.

Tómese esto únicamente como un ejemplo sencillo; probablemente la función *stable_sort()* sea muy similar, pero con un umbral mejor elegido.

Búsqueda binaria

Estos algoritmos representan distintas versiones de la búsqueda binaria. Todos reciben un rango ordenado de iteradores monodireccionales y un elemento a buscar.

binary_search()

Devuelve *true* si se encuentra un elemento equivalente en el rango y *false* en caso contrario.

lower_bound()

Devuelve un iterador que indica la primera posición donde se puede insertar el elemento manteniendo el orden. Por lo tanto, si existen elementos equivalentes al dado, el iterador indicará el primero de ellos.

upper_bound()

Devuelve un iterador que indica la última posición donde se puede insertar el elemento manteniendo el orden. Por lo tanto, si existen elementos equivalentes al dado, el iterador indicará *la posición posterior* al último de ellos.

equal_range()

Devuelve un par de iteradores; el primero de ellos es el que devolvería *lower_bound()* y el segundo, el que devolvería *upper_bound()*. Por lo tanto, el rango determinado por ambos iteradores contiene a todos los elementos equivalentes al dado. Si no existe ningún elemento equivalente, tal rango estará vacío.

Todos estos algoritmos realizan $O(\log n)$ comparaciones. No obstante, pueden realizar $O(n)$ operaciones elementales de iteración si los iteradores no son de acceso directo. El algoritmo *equal_range()* puede llegar a emplear el doble de comparaciones que los restantes.

Operaciones conjuntistas

Los siguientes algoritmos trabajan con rangos que representan conjuntos. Se exige que los rangos estén ordenados para poder asegurar algoritmos de la máxima eficiencia. En realidad, estos algoritmos generalizan las operaciones habituales sobre conjuntos permitiendo la existencia de repeticiones (elementos equivalentes).

Todos reciben dos rangos de iteradores de entrada de longitudes m y n. Salvo *include()*, reciben como último parámetro un iterador de salida en el que se copiará el resultado y que se devolverá.

include()

Devuelve *true* si el primer rango es subconjunto del segundo y *false* en caso contrario.

set_union()

Calcula la unión de los dos rangos de entrada.

set_intersection()

Calcula la intersección de los dos rangos de entrada.

set_difference()

Calcula la diferencia de los dos rangos de entrada.

set_symmetric_difference()

Calcula la diferencia simétrica (la unión de la diferencia del primero menos el segundo y de la del segundo menos el primero) de los dos rangos de entrada.

La complejidad de todos estos algoritmos es lineal. Se realizan, a lo sumo, $2 \cdot (m + n) - 1$ comparaciones (ninguna si ambos rangos están vacíos).

7.8.5. Montículos

Curiosamente no existe una clase contenedora ni adaptadora para representar montículos; sin embargo, STL posee algoritmos para cada una de las operaciones fundamentales de tratamiento de montículos (creación, inserción, eliminación del máximo y ordenación).

No obstante, la implementación del adaptador de secuencia *priority_queue* suele realizarse normalmente mediante el empleo de montículos.

Los algoritmos de tratamiento de montículos reciben únicamente un rango de iteradores de acceso directo como primer y segundo parámetro. Todos poseen dos versiones: la primera emplea para la comparación *operator* < y la segunda un predicado suministrado como tercer parámetro.

make_heap()

Construye un montículo en, a lo sumo, $3 \cdot n$ comparaciones.

push_heap()

Inserta el último elemento del rango en el montículo representado por el resto del rango. En realidad, restaura la propiedad del montículo en un árbol esencialmente completo donde ésta únicamente se «rompe», si acaso, entre el último elemento y su padre. Realiza, a lo sumo, $\log_2 n$ comparaciones.

pop_heap()

Elimina el máximo del montículo representado por el rango. En realidad, intercambia el primer elemento (el máximo) por el último y restaura la propiedad del montículo en el rango resultante de no considerar el ultimo elemento. Realiza, a lo sumo, $2 \cdot \log_2 n$ comparaciones.

sort_heap()

Ordena un rango con estructura de montículo. La ordenación no es estable. Realiza, a lo sumo, $n \cdot \log_2 n$ comparaciones.

Ejemplo:

El siguiente listado muestra cómo se puede programar el algoritmo de ordenación por montículo de Williams (*heapsort*) empleando *make_heap()* y *sort_heap()*.

monticulos/ordenacion.h

```
1  #include <algorithm>
2
3  template <typename Iterador> inline
4  void ordenacion_por_monticulo(Iterador principio, Iterador fin)
5  {
6    using namespace std;
7    make_heap(principio, fin);
8    sort_heap(principio, fin);
9  }
```

A continuación se muestran ejemplos de utilización de todas las funciones relacionadas con los monticulos.

monticulos/prueba.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <vector>
4  #include <algorithm>
5  #include <cstdlib>
6  #include "es-monticulo.h"
7  using namespace std;
8
9  int opcion();
10
11 int main()
```

```
12  {
13    vector<int> v;
14    while (true) {
15      cout << "v = [ ";
16      copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
17      cout << "]\n\n";
18      switch (opcion()) {
19      case 1:
20        make_heap(v.begin(), v.end());
21        break;
22      case 2:
23        if (es_monticulo(v)) {
24          v.push_back(rand() % 10);
25          push_heap(v.begin(), v.end());
26        }
27        else
28          cerr << "Se requiere un montículo\a" << endl;
29        break;
30      case 3:
31        if (es_monticulo(v) && !v.empty()) {
32          pop_heap(v.begin(), v.end());
33          v.pop_back();
34        }
35        else
36          cerr << "Se requiere un montículo no vacío\a" << endl;
37        break;
38      case 4:
39        if (es_monticulo(v))
40          sort_heap(v.begin(), v.end());
41        else
42          cerr << "Se requiere un montículo\a" << endl;
43        break;
44      case 5:
45        return 0;
46      default:
47        cerr << "Opción incorrecta\a" << endl;
48      }
49    }
50  }
51
52  int opcion()
53  {
54    cout << "Seleccione una opción:\n\n"
55            "1. Construcción de un montículo\n"
56            "2. Inserción en un montículo\n"
57            "3. Eliminación del máximo de un montículo\n"
58            "4. Ordenación de un montículo\n"
59            "5. Terminar\n"
60         << endl;
```

```
61    string opcion;
62    getline(cin, opcion);
63    return atoi(opcion.c_str());
64  }
```

Por último, y para completar, veamos la cabecera `"es-monticulo"`:

monticulos/es-monticulo.h

```
1   #ifndef ES_MONTICULO_
2   #define ES_MONTICULO_
3
4   #include <vector>
5
6   // Comprueba si un vector representa un montículo
7
8   template <typename T>
9   bool es_monticulo(const vector<T>& v)
10  {
11    using std::vector;
12    typedef typename vector<T>::size_type Indice;
13    const Indice n = v.size();
14    for (Indice padre = 0, hijo = 1; hijo < n; ++hijo) {
15      if (v[padre] < v[hijo])
16        return false;
17      if (hijo % 2 == 0)
18        ++padre;
19    }
20    return true;
21  }
22
23  #endif
```

Ejercicios

E7.1. Programe el algoritmo de ordenación por intercambio directo (método de la burbuja) para trabajar con vectores paramétricos. La ordenación se realizará ascendentemente respecto del *operator* < del tipo parámetro.

Escriba también un pequeño programa de prueba que solicite un tamaño, n, genere un vector de n enteros aleatorios distribuidos en el intervalo $[0, n)$ y lo ordene.

E7.2. Rehaga el ejercicio enterior, pero esta vez empleando iteradores de acceso directo. Así el algoritmo recibirá un rango sobre el que trabajar, en vez de un contenedor concreto.

E7.3. En el ejercicio anterior, al emplear iteradores de acceso directo se están imponiendo condiciones muy fuertes sobre los contenedores con los que el algoritmo puede trabajar. Por ejemplo, no podría emplearse con listas, ya que éstas no proporcionan iteradores de acceso directo.

No obstante, este algoritmo es lo suficientemente sencillo como para programarlo empleando iteradores bidireccionales sin reducir su eficiencia. Hágalo. Escriba también un programa de prueba, pero esta vez empleando listas.

E7.4. Uno de los aspectos clave que posemos encontrar en STL es su compromiso con la eficiencia. Compare a simple vista el tiempo que emplea la función *list<int>::sort()* en ordenar una lista de, por ejemplo, 10000 enteros, con el que tarda el programa de prueba del ejercicio anterior en ordenar esa misma lista. En conclusión, ¿merece la pena programar un algoritmo tal?

E7.5. Cree un programa que abra un fichero de texto cuyo nombre se le pase como parámetro y que lo lea carácter a carácter, poniendo cada uno en un *set<char>* creado a tal efecto. Muestre en la salida estándar el resultado y observe la organización.

E7.6. Cree un programa que abra un fichero de texto cuyo nombre se le pase como parámetro y que lo lea palabra a palabra. Si no se le da, leerá de la entrada estándar. Mostrará luego en la salida estándar la lista ordenada de las palabras, sin tener en cuenta la caja tipográfica (es decir, las mayúsculas y las minúsculas se considerarán equivalentes), que tengan más de un cierto número de caracteres, indicados por una opción del programa; si no se da, tomará 4.

E7.7. Cree, empleando la clase *Gna* del ejemplo, otra clase de objetos función binarios y adaptables para generar números pseudoaleatorios uniformemente distribuidos en el intervalo discreto $[a, b]$.

E7.8. Cree un programa que muestre las primeras sumas parciales de la *serie armónica* (es decir, los valores $H_n = \sum_{i=1}^{n} \frac{1}{i}$). Para ello, cree una clase de objetos función que genere sus términos y emplee *generate()* y *partial_sum()*.

E7.9. Escriba un algoritmo para seleccionar de un vector paramétrico el k-ésimo menor elemento con un tiempo promedio en $\Theta(n)$.

PISTA: Emplee *nth_element()*.

E7.10. ¿Qué pequeña modificación hay que hacer al algoritmo del ejercicio anterior para que calcule el k-ésimo mayor elemento?

E7.11. Escriba un programa que sea capaz de leer una cadena de ADN convenientemente representada y diga si es errónea, es decir, si contiene un par de nucleótidos cuyas bases no son complementarias. Debe mostrar el par erróneo.

Recuerde que los cuatro nucleótidos que forman el ADN contienen las bases adenina (A), guanina (G), citosina (C) y timina (T). Dado que en el ADN la adenina se empareja sólo con la timina y la citosina sólo con la guanina, cada cadena del ADN puede ser empleada como molde para fabricar su complementaria.

PISTA: Emplee *mismatch()* con un predicado apropiado.

E7.12. Escriba una clase *Naipe* para representar las cartas de una baraja española. Cree un programa sencillo que le permita jugar una partida en solitario a «las veintiuna».

PISTA: Emplee una cola doble de naipes para representar la baraja. Esto le permitirá extraer el de arriba y volverlo a introducir por debajo. Para barajar al principio de la partida emplee *random_shuffle()* (pásele un *Gna* construido con una semilla dependiente del tiempo).

Apéndice A

El código ISO-8859-1

El código ISO-Latin1										
Dígitos Izda./Dcha.	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	$	%	&	’
4	(	)	∗	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	ˆ	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	˜	DEL	PAD	HOP
13	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD
14	PLU	RI	SS2	SS3	DCS	PU1	PU2	STS	CCH	MW
15	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
16	NBSP	¡	¢	£	¤	¥	¦	§	¨	©
17	ª	«	¬	-	®	¯	°	±	^{2}	^{3}
18	´	µ	¶	·	¸	^{1}	º	»	$\frac{1}{4}$	$\frac{1}{2}$
19	$\frac{3}{4}$	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
20	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ
21	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û
22	Ü	Ý	Þ	ß	à	á	â	ã	ä	å
23	æ	ç	è	é	ê	ë	ì	í	î	ï
24	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù
25	ú	û	ü	ý	þ	ÿ				

Notas

1. La primera mitad de esta tabla, que va desde el 0 (NUL) hasta el 127 (DEL), coincide con el código conocido por los nombres ISO-646-IRV1991, ISO-646US, ISO-IR6, USASCII, y sobre todo, ASCII.

2. Toda esta tabla es la primera parte del código ISO-10646, también conocido como Unicode.

3. El código ISO-646US basta para representar todos los caracteres del latín y del idioma inglés; el ISO-8859-1 completo sirve, además, para los siguientes idiomas: afrikaans, alemán, catalán, castellano, danés, escocés, feroés, finés, francés, gaélico, gallego, islandés, italiano, neerlandés, noruego, portugués, sueco y vascuence.

4. Los códigos estándares de la serie ISO-8859 (de 8 *bits*) son:

ISO 8859-1	Lenguajes de Europa Occidental y América (Latin1)
ISO 8859-2	Lenguajes de Europa Oriental (Latin2)
ISO 8859-3	Lenguajes de Europa del sudeste y otros (Latin3)
ISO 8859-4	Lenguajes escandinavos y de los Balcanes (Latin4)
ISO 8859-5	Latín y cirílico
ISO 8859-6	Latín y árabe
ISO 8859-7	Latín y griego
ISO 8859-8	Latín y hebreo
ISO 8859-9	Modificación del ISO-Latin1 para el turco (Latin5)
ISO 8859-10	Lapón, lenguajes nórdicos y esquimal (Latin6)
ISO 8859-15	Lenguajes de Europa Occidental y América (Latin9)

   A diferencia del ISO 8859-1, el ISO 8859-15 contiene el símbolo del euro y proporciona una mejor codificación para el finés y el francés.

5. Los caracteres desde el 0 (NUL) hasta el 31 (US) y desde el 127 (DEL) hasta el 159 (APC) son caracteres de control. El carácter 32 «imprime» un espacio en blanco.

6. Las *secuencias de escape* corresponden a los siguientes códigos:

Sec. esc.	Mnemónico	Cód.	Descripción
`\0`	NUL	0	Terminador de cadenas
`\a`	BEL	7	Alerta, campana
`\b`	BS	8	Espacio atrás
`\t`	HT	9	Tabulador horizontal
`\n`	LF	10	Nueva línea (NL)
`\v`	VT	11	Tabulador vertical
`\f`	FF	12	Salto de página
`\r`	CR	13	Retorno de carro
`\e`	ESC	27	Escape (**¡esto no es estándar!**)
`\"`	"	34	Comillas dobles
`\'`	'	39	Comilla simple, apóstrofo
`\?`	?	63	Cierre de interrogación

Colofón

La confección de este libro nos ha costado tanto trabajo como poco dinero, en lo que se refiere a la adquisición de programas de procesamiento de texto y el entorno operativo. Ha sido compuesto en LaTeX empleando la distribución teTeX en sistemas GNU/Linux.

Para la escritura utilizamos el editor GNU Emacs con el paquete AUCTeX. El resultado se ha traducido a PostScript mediante el programa dvips de la mencionada distribución.

Todos los programas han sido probados empleando el compilador GNU C++ producido por la FSF (*Free Software Foundation*, Fundación del *Software* Libre).

Bibliografía

[1] Aburruzaga García, Gerardo; Medina Bulo, Inmaculada y Palomo Lozano, Francisco. *Por Fin: C ISO. Un Curso de C Estándar.* Servicio de Publicaciones. Universidad de Cádiz. 1998.

[2] Aburruzaga García, Gerardo; Medina Bulo, Inmaculada y Palomo Lozano, Francisco. *La Biblioteca Estándar de C.* Servicio de Publicaciones. Universidad de Cádiz. 1998.

[3] Austern, Matthew H. *Generic Programming and The STL. Using and Extending the C++ Standard Template Library.* Addison-Wesley. 1999.

[4] Horstmann, Cay S. y Budd, Timothy A. *Big C++.* Wiley. 2ª ed. 2008.

[5] Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley. 1999.

[6] Kernighan, Brian y Ritchie, Dennis. *The C Programming Language.* Prentice-Hall. 2ª ed. 1988.

[7] Musser, David R. y Saini, Atul. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library.* Addison-Wesley. 2ª ed. 2001.

[8] Plauger, P. J. *The Standard C Library.* Prentice-Hall. 1992.

[9] Stroustrup, Bjarne. *The C++ Programming Language.* Addison-Wesley. Special edition. 2000.

[10] Stroustrup, Bjarne. *Programming: Principles and Practice Using C++.* Addison-Wesley. 2008.

[11] Tondo, Clovis L. y Gimpel, Scott E. *The C Answer Book. Solutions to the Exercises in The C Programming Language, second edition.* Prentice-Hall. 3ª ed. 1989.

[12] Vandevoorde, David. *C++ Solutions. Companion to the C++ Programming Language, Third Edition.* Addison-Wesley. 1998.

Índice alfabético

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X