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.

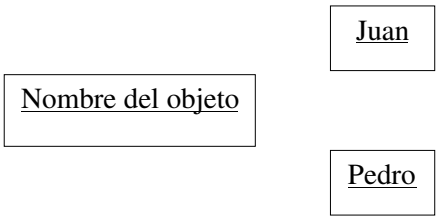

Figura 1.2: Representación de un objeto

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.

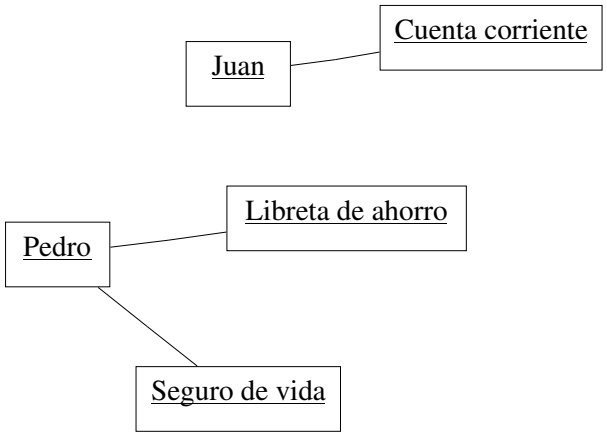

Figura 1.3: Representación de objetos relacionados entre sí

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

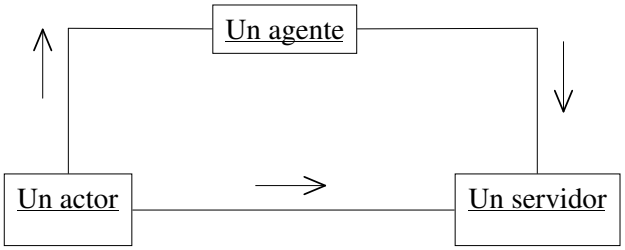

Figura 1.4: Categorías de objetos según su comportamiento

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.

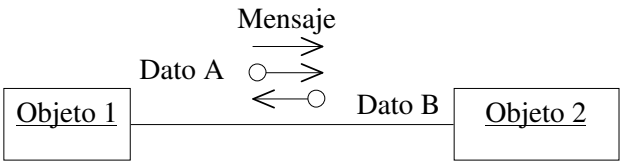

Figura 1.5: Representación de los flujos de control y de datos

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.

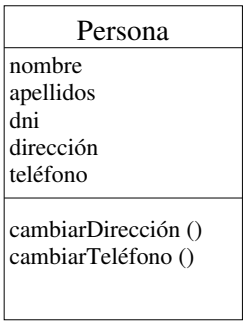

Figura 1.6: Representación de una clase

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.

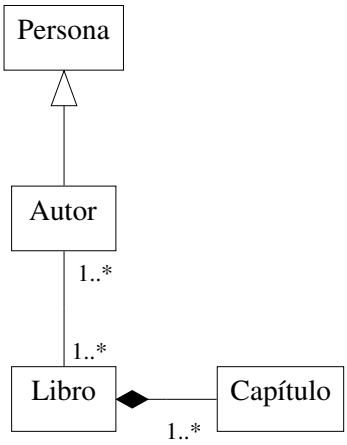

Figura 1.7: Relaciones entre clases

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.

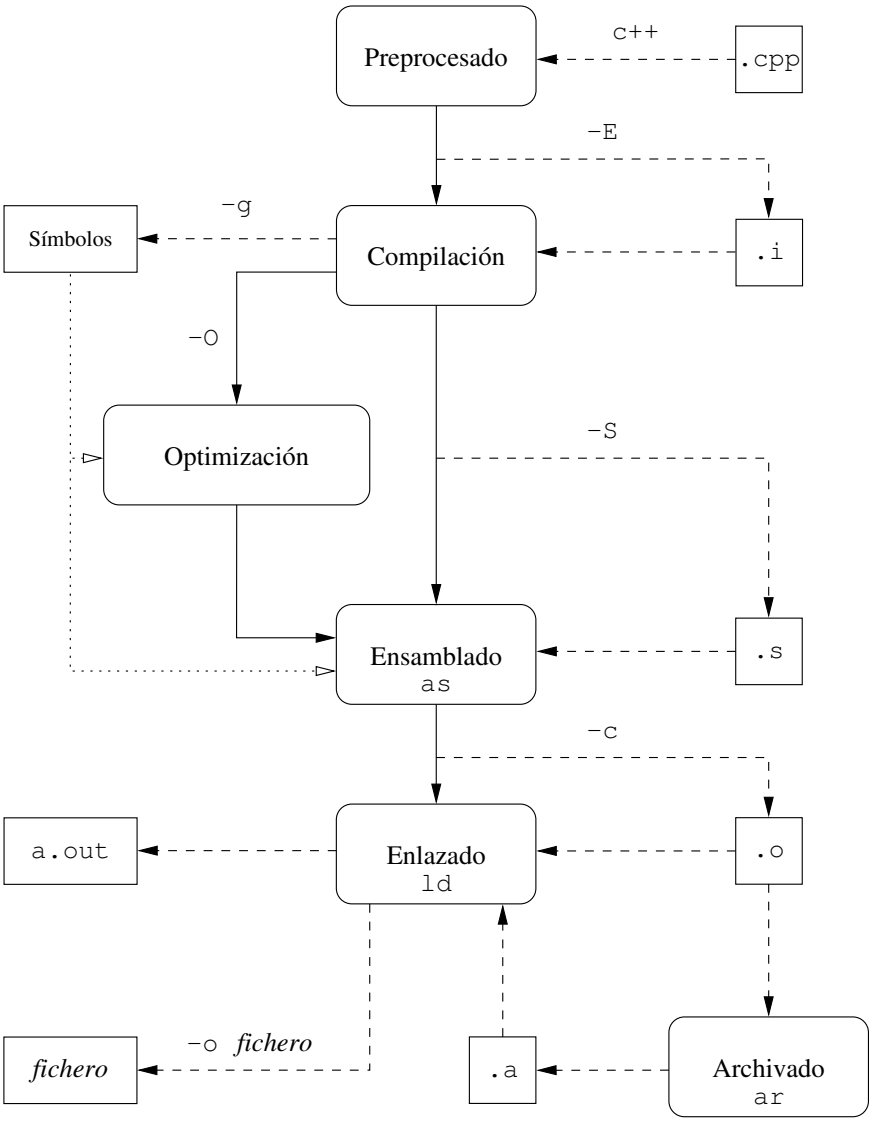

Figura 1.8: Pasos en la traducción de un programa

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.