C++/OOP

UN ENFOQUE PRÁCTICO

RICARDO DEVIS BOTELLA

1

INTRODUCCIÓN

Es costumbre que las primeras líneas de un texto procuren, de alguna manera, al lector una suerte de explicación del talante y ánimo del autor al escribir la obra que tiene ante sí. Bien: he aquí un enésimo libro sobre C++ y -¿cómo no?- sobre Programación Orientada a Objetos. De acuerdo, pensará el lector, pero ¿por qué éste y no otro?; o mejor, ¿qué tiene de especial el presente texto? Y la respuesta es... ¡una intención eminentemente didáctica! Lo que se pretende es introducir al lector en los esquemas básicos de la programación orientada-a-objetos -que en adelante llamaremos OOP- a través del uso de un lenguaje de amplia aceptación industrial, cual es C++. La aproximación será, sobre todo, práctica: procuraré no perderme en la maraña de siglas y conceptos que pueblan esta metodología y que frecuentemente desaniman al principiante, de forma que, tras el inevitable discurso teórico, siempre se buscará la aplicación concreta de lo expuesto mediante código en C++. Se trata, pues, de una introducción al lenguaje C++, pero, atención, utilizando de forma inseparable las técnicas y conceptos de OOP. El texto también quiere ser, por fin, ameno y, en lo posible, divertido: la tradición norteamericana de obras en las que el rigor no está reñido con un cierto humor, en ocasiones salvaje, será aquí observada con cierta complacencia. El tono será, pues, desenfadado pero exacto: a veces elemental, a veces no tanto.

¿A QUIÉN VA DIRIGIDO ESTE LIBRO?

No debemos engañarnos: las técnicas de OOP (y por tanto de C++, al que ya desde ahora deberemos acostumbrarnos a considerar como bien distinto de C) son difíciles de asimilar. Insisto: no sólo nos encontramos ante un área compleja, sino prolija, con abundancia de estándares y sumida en un continuo cambio evolutivo. Las experiencias en U.S.A. indican que los estudiantes tardan de seis a nueve meses (si no más) en asimilar verdaderamente y poner en práctica de forma efectiva los conceptos, técnicas y metodologías aprendidas. Así que debo suponer que el lector tendrá conocimientos de algún lenguaje estructurado, como Fortran, Pascal, C, etc. Dado que pretendemos trabajar en C++ y que este lenguaje comparte muchas de las bases de C, sería deseable que el lector conociera al menos las bases del lenguaje C, aunque lo ideal sería tener alguna experiencia en ANSI C. De cualquier forma existe una gran profusión de textos didácticos sobre C y ANSI C, por lo que se obviarán las explicaciones sobre las construcciones en tales lenguajes.

¿QUÉ MATERIAL SE NECESITA?

Debo insistir en un tópico: sólo programando se aprende a programar. Y esto es aún más cierto, si cabe, en C++. El lector deberá contar con un compilador que le permita chequear el código escrito: muchos programadores de C se quedarían asombrados al ver la larguísima letanía de errores y *warnings* que aparecerían al compilar como C++ su código C. La creciente complejidad de las implementaciones C++ exige cada vez más requerimientos hardware. El lector necesitará, pues, de un compilador que soporte, preferiblemente, la versión 3.0 del AT&T C++, a la vez que máquina suficiente para soportarlo, junto con las pertinentes librerías de clases, como más adelante veremos.

¿QUÉ OBJETIVO SE PERSIGUE?

Sorprende que en la iniciación a cualquiera de los tópicos de OOP el principiante siempre se encuentre con introducciones, preámbulos e incluso introducciones de introducciones. Bien, esto es desafortunadamente irremediable: es necesario cambiar muchas cosas (entre ellas la "forma de pensar" del programador) para poder aplicar eficientemente uno o dos conceptos clave. Dado que se supone, sobre todo en C++ y Object Pascal, que el interesado posee conocimientos previos de programación estructurada, buena parte del tiempo se emplea repitiendo: ¡olvídese de cómo lo estaba haciendo: piense en objetos! Esto es exactamente, pues, lo que se pretende en este libro: sumerger al lector en un nuevo lenguaje (C++) pero siempre desde el punto de vista de la OOP, lo que ayudaría a que profundizara más tarde en los tópicos introducidos mediante el uso de algunos de tantos excelentes textos sobre el tema. Se pretende, por tanto,

guiar al principiante entre el oscurantismo y la verdadera complejidad de un nuevo lenguaje y un novedoso (sólo para él, por supuesto) paradigma: la Programación Orientada-a-Objetos. Piénsese que un no muy extenso detalle sobre, por ejemplo, la característica de *templates* (plantillas) de C++ ocupa la mayor parte de un magnífico texto de Robert Murray sobre el lenguaje. Mi objetivo es modesto: tras el último capítulo del libro (o quizás afortunadamente antes) el lector debería ser capaz de desarrollar programas relativamente simples en C++ que funcionaran bajo Microsoft Windows 3.1, OSF/Motif, OS/2 2.1, MS Windows NT y Mac utilizando librerías comerciales de clases (como, por ejemplo, Tools.h++, Codebase++) y entornos de aplicación (como ObjectWindows ó C++/Views).

BIBLIOGRAFÍA DISPONIBLE SOBRE C++

Lamentablemente existen contados libros originales en castellano sobre C++ y, por lo que yo conozco, actualmente se reducen a la introducción elemental al tema escrita por Francisco Javier Ceballos ("Introducción a C++ y a la Programación Orientada al Objeto"), referida básicamente a la versión 1.2 de C++ (con destellos de la versión 2.0), lo cual, teniendo en cuenta que actualmente se trabaja en base al cfront 3.0 de AT&T, la convierte en un tanto desfasada; y al buen texto "Programación en C++", de los hermanos Enrique y José Hernández Orallo, ajustado a AT&T 3.0 e incorporando "plantillas" y manejo de excepciones. Naturalmente no cuento aquí con los típicos manuales de compiladores del tipo "Cómo manejar Borland C++ 4.0 en 1000 días", por razones obvias, así como tampoco con las traducciones de las obras inglesas que, a poco, se irán introduciendo en el mercado. Los manuales de la inmensa mayoría de los compiladores comerciales de C++ y el grueso de la bibliografía están en inglés, y en U.S.A. se está produciendo un verdadero "boom" editorial con la OOP que, a poco, veremos en España. Como, insisto, el tema es difícil, pospondré la relación de material bibliográfico hasta el artículo final, en el que además reseñaré brevemente tanto los libros como los compiladores, librerías de clases y entornos de aplicación más interesantes.

EL PROBLEMA DE LAS VERSIONES

Esta cuestión es indicativa de lo que en OOP es tónica general: la falta de estandarización. Frecuentemente oiremos de las versiones 1.2, 2.0, 2.1 y 3.0 de C++. Existe, por otro lado, un comité ANSI dedicado a la estandarización del lenguaje y que no entiende de tales numeraciones. ¿Qué ocurre? Bueno, como C++ fue creado por el Dr. Bjarne Stroustrup, de los laboratorios AT&T Bell, y éstos siempre se han mantenido en la vanguardia del lenguaje, los compiladores comerciales de otras casas se han basado en la numeración de AT&T. El comité ANSI X3J16, creado para la estandarización de C++, admitió en su día, por otro lado, el texto "Manual de Referencia C++ Anotado" (que en adelante denominaremos ARM, como es práctica común en los textos americanos) del Dr. Stroustrup y Margaret Ellis como documento base del lenguaje, por lo que en puridad no cabe hablar de versiones del lenguaje. No hay que olvidar, no obstante, que el nombrado comité ANSI ni siquiera posee la cualificación de internacional: a pesar del interés de sus miembros, todavía está circunscrito al ámbito nacional estadounidense. El mercado, por otra parte, sigue básicamente la numeración de AT&T, criterio que por facilidad para distinguir entre distintas características del lenguaje yo también adoptaré en lo que sigue. Cabe destacar, al fin, que periódicamente el comité ANSI X3J16 publica un borrador del estado actual del estándar del lenguaje, y que se puede conseguir directamente de X3. Este borrador, que es una suerte de ARM ampliado y consensuado, está afortunadamente constituyéndose en fuente y modelo de los compiladores comerciales (como ocurre, por ejemplo, con Borland C++ 4.0).

ALGUNAS NOTAS SOBRE SIGLAS

Intimida abrir cualquier revista técnica sobre OOP por la cantidad de siglas y jerga incomprensible que aparece en cada página. Encontramos con demasiada facilidad claves como OBCS, HSC, ODS, EER, etc. de difícil traducción. ¿Qué ocurre? ¿Vive la comunidad OOP en un mundo aparte? Bien, la verdad es que sí. No existen técnicas estándares ni en análisis ni en diseño, por lo que investigadores y equipos privados desarrollan continuamente técnicas propias, frecuentemente sin ningún nexo común; junto con éstas, naturalmente, desarrollan también sus propias siglas y terminologías. En general podemos afirmar que O representa Objeto, OO equivale a Orientado-a-Objetos, A vale por Análisis y D por Diseño, R significa Requerimientos, L representa Lenguaje y C suele equivaler a Clase. De esta forma OOAR, por ejemplo, significa Requerimientos del Análisis Orientado-a-Objetos. Mi consejo es, de cualquier manera, que el lector no dé por asumidos significados intuitivos a siglas o conceptos cuya procedencia no conozca, por muy elementales que éstos parezcan.

BREVÍSIMA HISTORIA DE C++

Básicamente C++ es el resultado del acoplamiento al lenguaje C de algunas (no todas) de las características de la OOP. Su creador, el hoy famoso Bjarne Stroustrup, trabajó desde 1980 en lo que por aquel entonces se denominaba "C con clases", directamente proveniente de una cierta simbiosis con el lenguaje SIMULA. La publicación de varios ensayos sobre el tema culminó en 1986 con la publicación por el Dr. Stroustrup de la obra "El Lenguaje de Programación C++", en la que se establecieron las bases del lenguaje como hoy lo conocemos. Este texto ha representado para C++, aunque tal es pura opinión, lo mismo que en su día representó para el lenguaje C el libro "El lenguaje de Programación C" de Kernighan & Ritchie. Actualmente se encuentra disponible la segunda edición de aquél texto, en la que se detalla la versión 3.0 del lenguaje.

¿ES C++ UNA MERA EXTENSIÓN DE C?

El lector ya puede imaginarse que la respuesta es NO. Lo que puede que le sorprenda es hasta qué punto son diferentes ambos lenguajes. Es frecuente encontrarse con la afirmación "C++ es un superconjunto de C", lo que sugiere que todo el código C podría compilarse como C++. En realidad esto no funciona: únicamente un subconjunto de C, conocido como C-, cumple esta condición. Usar C++ como C puede ser, por otra parte, un error en algunos casos. El lenguaje C, por su tremenda flexibilidad, es idóneo para ser extendido con relativa facilidad: han surgido así derivaciones de C como C++, C//, Objective-C, etc., cada una con su propia idiosincrasia y normativa. Esto sugiere que C ha servido, en definitiva, únicamente como sustrato básico sobre el que desarrollar nuevas técnicas y metodologías. C++ es, de cualquier forma, un mejor C. De ahí proviene su nombre: C con el operador de post-autoincremento.

OBJETOS: UNA NUEVA FORMA DE PENSAR

El término "objeto" se muestra omnipresente en la literatura de OOP, y sin embargo es frecuente que el lector, tras leer diversas descripciones y definiciones, termine el texto aún más confundido que al empezarlo. Así, se puede leer -correctamente- que "un objeto es una encapsulación de datos y de los métodos para manipular a éstos", o, en un nivel superior, que "es la instancia de una clase, siendo ésta la entidad conceptual que encapsula los comportamientos comunes a los objetos que representa". ¿Qué ocurre? Que estas definiciones únicamente las entiende quien ya las entendía, pues al resto les representa una relación parecida a la que tienen, por ejemplo, las funciones de variable complejas con los procedimientos holográficos: o sea, suponen que es verdad pero no les dice nada, como ocurre en la célebre anécdota del economista.

¿Qué es un objeto? Bien, aquí ocurre lo que con los conceptos "grupo", "átomo" o "conjunto": la aplicación del concepto general depende del nivel de abstracción elegido. Para no perdernos intentaremos una técnica muy usada en OOP: el símil antropomórfico. Relacionaremos, así, objetos reales con el concepto en estudio. Antes de proseguir debemos, sin embargo, desechar algo: el lector deberá olvidar, y aun repudiar (en la mejor tradición bíblica), cualquier asociación de objetos en el mundo real con posibles técnicas de programación por él conocidas. O sea: no debe pensar como programador, ni siquiera en el más elemental de los niveles. No debemos asimilar, pues, los objetos como datos, ni como métodos o servicios. Dicho esto, detallemos algunos objetos del mundo real: una factura, una sala de cine, un libro, un programa, el sol, un avión, etc. El lector aquí podría decir: "bien, bien, esto son objetos, ya lo sé, pero ¿qué tiene esto que ver con la programación?". Paciencia, paciencia. Retengamos, por ahora, el esquema ilustrado por el siguiente ejemplo: si usamos, verbigracia, un objeto "lupa" con un objeto "sol" obtendremos el resultado de un rayo calórico que podría incidir sobre un objeto "papel" quemándolo. Aquí vemos distintos objetos interactuando entre sí, con la siguiente interpretación formalista: el "sol" envía un mensaje luminoso a la "lupa", y ésta envía un mensaje calórico al "papel", que contesta (en el mundo real de forma irremediable) con un mensaje a sí mismo que dice "quémate". Examinemos el mismo esquema desde otro enfoque: no es que existan unos datos puros que configuren los objetos y unos servicios o métodos etéreos y ajenos a los objetos y que los afecten. El "sol" no es un cúmulo o estructura de datos, como no lo son el "papel" o la "lupa". No existen, tampoco, métodos o funciones ajenos a tales objetos: que se queme el "papel" depende de las características intrínsecas de éste, pues no existe una "función de quemado" general. ¡Un momento! ¿No existe una noción general de "quemar"? Ciertamente: tan cierto como que la materia se compone de átomos, aunque tal conocimiento no nos sirva de mucho. Nosotros no queremos "quemar" una abstracción o un "objeto general": deseamos "quemar" un "papel" ¿y quién sabrá más de "quemar un papel" que el "papel" mismo? Intentemos otro ejemplo: ¿existe una función general en los seres humanos para "dormir" y que en los españoles se concreta además en la "siesta"? ¿O es que la función de dormir específica de los españoles incluye como individualidad la "siesta"? Parece que nuestra intuición se inclina por lo segundo.

Conservemos la intuición y abordemos un objeto bien familiar: una ventana de Microsoft Windows. Uno de tales objetos poseería unas determinadas características internas (barra de título, menú, color, scroll-bars, etc.), a la vez que unos servicios, métodos o funciones específicas (redimensionamiento, movimiento, iconización, etc.). En este contexto aparece más clara la comunicación entre objetos: un click del ratón puede originar un mensaje dirigido a la ventana que, a su vez, origine un mensaje de la ventana a ella misma forzando su maximización. Hemos puesto en el mismo saco a objetos, métodos, datos y mensajes. Bien, retengamos de momento esta amalgama de ideas.

PROGRAMACIÓN ORIENTADA-A-OBJETOS

Cabría aclarar, antes de nada, que prefiero la expresión Orientación-a-Objetos frente a la también muy usada Orientación-al-Objeto. Es una matización semántica en la traducción del término inglés, que pierde la riqueza contextual de la adjetivación en la traducción literal. Dicho esto, entremos en el núcleo del asunto: ¿qué demonios es la OOP (Object-Oriented Programming)?

La OOP es, simplificando, un sistema de programación que utiliza objetos para la resolución de problemas, interrelacionándolos mediante mensajes. Imaginemos, por ejemplo, que queremos desarrollar una aplicación para la gestión organizativa de un despacho profesional. En la vida real nos encontraríamos con directores, secretarias, máquinas de escribir, impresoras, etc. Una posible solución en OOP sería trasladar estos objetos al espacio de programación: el objeto teléfono, al sonar, lanzaría un mensaje primero al centro de control y luego a una determinada secretaria mecanizada, la cual, en calidad de objeto, podría responder reteniendo brevemente la llamada y enviando un mensaje de aviso a la operadora correcta. De nuevo vemos que no nos movemos con esquemas funcionales derivativos: no existe una función general de llamada telefónica (pueden existir teléfonos digitales, centralitas, etc.). Modelamos nuestra visión del problema a través de objetos, que se relacionan entre sí por pocos canales, de forma que la comunicación de mensajes se pretende alta y clara. En definitiva se pretende que las ligazones entre distintos objetos (o entidades) sean transparentes para el observador exterior. Naturalmente esto favorece sobremanera la comunicación entre los responsables de las distintas etapas de desarrollo de software.

Un ejemplo de uso habitual podría ser el de los gestores de bases de datos relacionales. Desde la óptica de la OOP los ficheros de tipo, por ejemplo, *.DBF, tan comunes en el mundo PC, se asocian a objetos "dataBaseFile", mientras que los índices (del tipo *.NDX ó *.MDX) se asocian a objetos del tipo "indexFile". La ordenación de una base de datos consistiría en un mensaje que un objeto "indexFile" dirigiría a un objeto "dataBaseFile" del tipo "ordénate en base al índice con etiqueta X". El habitual comando "open" se convierte, así, en un mensaje que se dirige al objeto "dataBaseFile", forzando la apertura del fichero de base de datos. Naturalmente la extensión lógica de estos conceptos nos llevaría a superar el concepto de bases de datos relacionales, entrando en lo que se ha dado en denominar ODBMS (Sistemas Gestores de Bases de Datos de Objetos), conceptualmente diferenciadas de aquéllas.

¿HA MUERTO LA PROGRAMACIÓN ESTRUCTURADA?

Definitivamente NO. Al menos no todavía. En realidad la OOP podría considerarse como una extensión natural de la programación estructurada, dado que, en definitiva, aquélla surgió debido a las carencias y debilidades de ésta. Permitámonos un breve repaso histórico: por allá por los años 70 una nueva metodología denominada "desarrollo estructurado", basada en la independencia de datos y funciones o métodos, permitió superar la en aquel entonces llamada "crisis del software". Así como el nombre de Timothy Leary quedó indeleblemente unido al movimiento hippie, el diseño "top-down" se convirtió en sinónimo de enfoque estructurado: un problema se asimila a una función o procedimiento, que se descompone en problemas más pequeños, que a su vez se descomponen en otras funciones más pequeñas, hasta llegar a problemas descomponibles. Gracias a estos métodos los sistemas software fueron poco a poco creciendo en tamaño y complejidad, de forma que al final los problemas solucionados por el desarrollo estructurado generaron nuevos problemas de más difícil solución. A principios de los años 80 las empresas recabaron en que buena parte de su presupuesto se dilapidaba en el mantenimiento de verdaderas monstruosidades software: las labores de depuración se volvían más costosas según iba aumentando el tamaño de los programas; además, cualquier modificación repercutía normalmente en todos los módulos, de manera que un programa bien chequeado pasaba a ser, después de una pequeña reforma, motivo de nuevos y largos testeos y validaciones de estabilidad. El desarrollo de librerías, potenciado sobremanera, había llegado a un punto crítico de ineficacia: si una función se ajusta exactamente a lo que queremos se usará; si no, habrá que codificarla de nuevo. Todo el tiempo que el equipo de desarrollo pudiera dedicar a la prototipación y pruebas de un sistema se perdía, por otra parte, inevitablemente, debido a las escasas posibilidades de reutilización del código.

El panorama no parece muy alentador. Para intentar salir de este círculo vicioso surgen en un primer momento lenguajes modulares como Ada, Modula-2, etc., que solucionan algunos de los problemas planteados. La situación es, sin embargo, todavía insatisfactoria, así que la lógica evolución de los conceptos de modularidad y reutilización del código origina un nuevo paradigma que se llamará OOP.

Hay que pensar, pues, que la OOP es un escalón muy avanzado en el desarrollo de la programación estructurada, pero no tan avanzado que le permita prescindir de ésta. De hecho muchos autores de la comunidad OOP opinan que habría que tender a un más féliz acercamiento entre ambas metodologías.

¿POR QUÉ C++?

¿Es C++ el mejor OOPL (Lenguaje de OOP)? ¿Qué tiene C++ que no tengan los demás OOPL's ó que no tenga C? ¿Es C++ un montaje de marketing por el que se nos intenta vender como nuevos los viejos esquemas, una vez remozados? En definitiva, ¿por qué C++ y no otro lenguaje?

Bien. Hemos visto en el apartado anterior que la OOP intenta solucionar los problemas originados por el "enfoque estructurado". Todo esto está muy bien, pero estas nuevas técnicas necesitan de lenguajes de programación adecuados. Surgieron, así, lenguajes de nueva creación como Smalltalk, Eiffel, Actor, etc. Por otro lado se intentó dotar de extensiones Orientadas-a-Objetos a los lenguajes clásicos más importantes, como C, Pascal, Fortran, Ada, Cobol, Lisp, etc., originando C++, Object Pascal, CLOS, etc.

Naturalmente hay que escoger: ventajas y desventajas. Los lenguajes Orientados-a-Objetos "puros" permiten una enorme flexibilidad, aunque casi siempre en detrimento de la eficacia. Pensemos que en Smalltalk, por ejemplo, no existen tipos predefinidos en el sentido que conocemos: el tipo es una mera equiqueta pegada a un objeto y que, en un momento dado, podemos cambiar, pegándole otra distinta. Así, por ejemplo, un dato cambiaría de tipo en el transcurso del programa: algo inpensable en C y aun en C++. Por otra parte, la ligadura de un mensaje dirigido a un objeto con una respuesta o método determinado se produce siempre, en Smalltalk, en tiempo de ejecución: es decir, no sabemos en tiempo de compilación qué cuerpo de una determinada "función" se ejecutará al ejecutar el programa (notemos que la misma "función" o "método" puede tener distintos cuerpos en distintos objetos). Naturalmente, como el lector inmediatamente habrá adivinado, esto origina no pocos problemas en la depuración y testeo de las aplicaciones, a la par que dificulta enormemente la captura de errores en tiempo de ejecución (pensemos en qué ocurre cuando dirigimos un mensaje a un objeto equivocado -o sea, no cualificado para responderlo-: ¡el desastre!). Los objetos, en Smalltalk, suelen, por tanto, incorporar su propio "depurador". Debo significar que en la actualidad Smalltalk está siendo muy usado en prototipación rápida y, aunque existen compiladores que permiten la creación de archivos ejecutables (como Smalltalk/V de Digitalk), en general los entornos Smalltalk funcionan como intérpretes, con la pérdida de eficacia que esto supone en el plano comercial.

Un nivel aceptable de compromiso es el proporcionado por el lenguaje "puro" Eiffel, creado (o más bien publicitado) por el Doctor Bertrand Meyer en 1.988 y recientemente adoptado para distintos proyectos por la Agencia Espacial Europea. Este lenguaje enfatiza, entre otras interesantísimas características, el uso de PRECONDICIONES y POSTCONDICIONES, algo que poco a poco se ha ido incorporando a C++, como se puede observar en muchas de las librerías comerciales actuales.

C++, en contra de lo expuesto y como ya se comentó anteriormente, es un lenguaje híbrido: ha adoptado cuantas características de OOP no perjudicaran su efectividad. Así, por ejemplo, la ligadura dinámica de un mensaje a un método se ha implementado (mediante las denominadas funciones virtuales) de forma que soporta un chequeo de tipos en tiempo de compilación. Se tiene, por un lado, una mejora sustancial de las capacidades de C, a la vez que se obtiene buena parte de lo mejor de la OOP sin perder aquellos beneficios. Es lo mejor de dos mundos, como suele decirse. Un poco más vehemente, el Doctor Stroustrup ha llegado a afirmar que C++ es un lenguaje destinado a "los programadores serios". No quiere esto decir, sin embargo, que otros lenguajes no sean "serios", sino que éste difícilmente podría ser utilizado por programadores aficionados o de fin de semana

Retomando la pregunta inicial, vayamos al grano: escogemos C++, entre otros lenguajes, por sus características conjuntas de robustez, eficacia y flexibilidad. El mercado industrial ha efectuado ya, por otra parte, su decisión y ha convertido a C++ en el estándar industrial de facto en OOP, con todo lo que esto conlleva (pensemos, por ejemplo, en el arrinconamiento comercial del sistema de video Beta, posiblemente mejor que el VHS pero rotundo perdedor en el mercado ante éste). Así, poco a poco, infinidad de firmas han ido desarrollando extensiones, librerías, compiladores, entornos de aplicación y herramientas para C++.

¿ES LA OOP EL VÉRTICE DE LA PROGRAMACIÓN?

¿Hemos llegado al límite de las posibilidades de los sistemas de programación? O de otra forma: ¿Es la OOP el no va más del sumum? Bien, sólo un fanático podría contestar afirmativamente (y el autor podría hablar durante horas de este tipo de cerrazón mental). La OOP es únicamente un estadio en la evolución de las técnicas de programación, aunque quizá en estos momentos sea el estadio más avanzado. Pero en informática, aún más rápidamente que en otros campos, no existen metodologías inamovibles, sino únicamente peldaños en una de tantas escaleras. Tal vez la idea de los vasos comunicados fuera un buen símil: distintas tecnologías despuntan individualmente en una primera fase para inmediatamente después converger en metodologías conjuntas. Veamos, por ejemplo, lo que ha sucedido con las técnicas CASE, que en su momento parecieron conducir los esquemas de desarrollo soft a un punto de no-retorno. Últimamente la comercialización de dichas herramientas se ha reducido de forma drástica: la eclosión de la OOP y de los entornos gráficos, entre otras razones, ha causado una cierta suspensión, hace pocos años impensable, de la metodología CASE. ¿Qué es lo que, en definitiva, ha sucedido después? Las técnicas CASE y de OOP están lentamente uniéndose: ya se habla de herramientas CASE con extensiones OOP, o aun de entornos de aplicación OOP con utillería CASE, apareciendo, así, herramientas OO-I-CASE, OOCASE, etc. Al final la practicidad se une a la brevedad: "lo mejor" incrementalmente se fusiona con "lo bueno", y la industria sigue avanzando.

2

OOP: CONCEPTOS BÁSICOS

Examinadas, aunque sucintamente, las ideas que subyacen bajo el nuevo "paradigma de objetos", vamos a abordar los conceptos que formalmente definen una metodología como Orientada-a-Objetos. Un sistema se califica como Orientado-a-Objetos cuando reúne las características[1] de: *abstracción*, *encapsulación*, *herencia* y *polimorfismo*. Antes de abordar estas cualidades repasaremos, no obstante, los conceptos básicos que las informan: *objetos*, *mensajes*, *clases*, *instancias* y *métodos*. Debo hacer hincapié, de nuevo, en que tales conceptos deberán ser considerados por el lector con un "espíritu vacío de prejuicios" y sin pensar, en esta primera fase, en su inmediata aplicación a la programación. Una advertencia al lector: conozco a algunas personas que no han podido pasar de ser 'lectores de introducciones', porque éstas les dejan en un estado similar al que tenían antes de leerlas. Es difícil conjugar facilidad de lectura y amenidad con rigurosidad y formalismo práctico. Indudablemente, por otro lado, conforme vayamos avanzando (porque lo que aquí se pretende es, ante todo, avanzar) los conceptos se tornarán más complejos y dependerán en buena medida de lo que hayamos visto anteriormente. No hay más remedio, entonces que resignarse y releer, releer como norma, pues, como ya ha quedado dicho, C++ es un lenguaje díficil para programadores serios. Entremos a saco, sin más preámbulos, en la esencia teórica de la OOP, con el fardo de terminología que esto conlleva, atacando, así, la parte "correosa" del tema.

OOP: CONCEPTOS TEÓRICOS BÁSICOS

[1] Lo cierto es que a estas alturas subsisten importantes diferencias de criterio entre distintos autores a la hora de establecer los pilares en que se apoya la Programación Orientada-a-Objetos. El mismo autor del presente texto sostiene, por ejemplo, que la herencia no es una característica básica del paradigma, sino más bien un mecanismo que permite la implementación de jerarquías polimórficas. Intentanto ser lo más general posible, he reflejado, no obstante, las características más comunmente aceptadas por el grueso de expertos en estas áreas.

Un **objeto** es una encapsulación abstracta de información, junto con los métodos o procedimientos para manipularla. Según la esquemática definición de Wegner, "un objeto contiene operaciones que definen su comportamiento y variables que definen su estado entre las llamadas a las operaciones". Bueno, pensemos en un ejemplo asequible: una sala de cine. Imaginemos un objeto del tipo "sala de cine" donde los datos, variables o información estarían constituidos por los espectadores; imaginemos también (y esto es importante) que los espectadores no pudieran moverse por sí solos dentro de la sala (como si estuvieran catatónicos). ¿Quiénes serían los manipuladores, métodos, procedimientos u operaciones encargados de manipular a los espectadores? Indudablemente los acomodadores de la sala, aunque también el personal directivo de la misma.

Un **mensaje** representa una acción a tomar por un determinado objeto. En el ejemplo anterior, la orden "que comience la proyección de la película" podría ser un mensaje dirigido a un objeto del tipo "sala de cine". Otro mensaje podría ser la orden de "desalojo de la sala" en funciones no-continuas. Notemos que el mensaje se refiere únicamente a la orden, y no tiene que ver con la forma como ésta es respondida. En C++ un mensaje equivale al PROTOTIPO de una función miembro en la descripción de una clase. O sea, si pensamos en una posible función "especialmente restringida a un objeto" (que en C++ se denomina función miembro) tal como "empezarProyeccion", el mensaje estaría representado por el prototipo de tal función, y no por su definición o implementación específica.

Una **clase** equivale a la generalización o abstracción de un tipo específico de objetos. Los polígonos con tres lados iguales podrían ser generalizados, por ejemplo, en una clase que llamaremos "trianguloEquilatero". Hablar de clase es, así, hablar de una determinada clase de objetos. En C++ una clase ("class", según la terminología del lenguaje) es un tipo definido-por-el-usuario. De esta forma, y siguiendo con el ejemplo cinematográfico, el objeto abstracto "Sala de Cine" sería implementado como la clase "SalaDeCine" en C++, habiendo definido así un nuevo tipo de dato abstracto, que será manejado por el compilador de parecida forma a como lo hace con los tipos predefinidos (int, char, float, etc.). Podremos, así, como ya veremos, declarar un determinado objeto del tipo (o de la clase) "SalaDeCine".

Una **instancia** es la concrección de una clase. Una instancia de la clase "SalaDeCine" sería, por ejemplo, el objeto "cineParadox". El concepto de instancia, en realidad, une la noción de objeto con la de clase. Esto quiere decir que en el esquema de OOP, a semejanza de lo que ocurre en el mundo real (aunque normalmente de forma inadvertida), debemos identificar en primer lugar una abstracción general, delimitar las características comunes de un grupo de objetos, y luego poner nombre (o identificar especialmente) a uno o más de ellos. Tomemos, verbigracia, el número "1/7": lo primero que se nos ocurre es que tal es un número "racional". En OOP diríamos que es un objeto o instancia de una clase de objetos denominada "Racional", cuyas características serían las que conocemos (a nivel de programación) de tal conjunto.

Un **método** consiste en la implementación en una clase de un protocolo de respuesta a los mensajes dirigidos a los objetos de la misma. La respuesta a tales mensajes puede incluir el envío por el método de mensajes al propio objeto y aun a otros, también como el cambio del estado interno del objeto. En C++ los métodos están implementados como DEFINICIONES de funciones miembro de una clase, representando el conjunto de mensajes al

que los objetos de tal clase pueden responder. Revisando el ejemplo que examinábamos al hablar de "mensajes", podemos decir que la respuesta al mensaje "empezar la Proyección" podría significarse en la serie de instrucciones tendentes a efectuar la acción de visionado de la película: apagar las luces, encender el proyector, etc. Estamos hablando, como el lector habrá adivinado, del cuerpo de la función miembro "empezarProyeccion".

ENVÍO DE MENSAJES A OBJETOS

Reexaminemos los conceptos intuitivos expuestos hasta ahora. Hemos visto que mientras que la programación estructurada se basa, sustancialmente, en llamadas de alto nivel a determinadas funciones y rutinas, la OOP consiste, básicamente, en las relaciones entre objetos (instancias de clases en C++) que responden a mensajes de acuerdo con los métodos (funciones miembro en C++) establecidos en el protocolo de descripción de sus respectivas clases. Bueno, esto se va complicando, pero es inevitable: no se puede evitar indefinidamente la endiablada terminología "de objetos". Y no hemos hecho más que empezar. Ok: prosigamos.

El "envío de un mensaje" a un objeto equivale, en C++, como ya he indicado, a una llamada a una función miembro de la clase correspondiente al objeto. En realidad las expresiones "métodos", "envío de mensajes", "instancias de variables", etc. pertenecen originariamente al lenguaje Smalltalk. Volvamos al ejemplo del cine y "lancemos mensajes":

```
SalaDeCine cineParadox;
SalaDeCine *punteroASalaDeCine;
```

Hemos declarado por un lado un objeto denominado cineParadox, del tipo definido-por-el-usuario SalaDeCine. Seguidamente hemos declarado un puntero al mismo tipo de dato abstracto. El mensaje "que comience la proyección de la película", dirigido a la sala "Cine Paradox", podría ser codificado así:

```
cineParadox.empezarProyeccion();
```

Esto es, el mensaje o llamada de función se ha dirigido directamente al objeto. Pero veamos el siguiente código:

```
punteroASalaDeCine = new SalaDeCine( cinePalafox );
punteroASalaDeCine->empezarProyeccion;
```

En este ejemplo primero dirigimos el puntero declarado anteriormente hacia un objeto de nueva creación, alojado en la memoria de almacenamiento libre mediante el operador **new** (podría decirse que una mejora sustancial sobre la conocidad función **malloc** de C), del tipo SalaDeCine, al que hemos denominado (como variable en el fondo que es) cinePalafox. Seguidamente el mensaje es dirigido al objeto cinePalafox por mediación de tal puntero al mismo.

Vemos, pues, que la notación '.' sirve para enviar mensajes directamente a un objeto, mientras que '->' se usa para el envío de mensajes a los objetos a través de punteros que apunten a los mismos.

Cabría notar aquí, redundando en lo apuntado anteriormente, que el prototipo de función empezarProyeccion() corresponde al mensaje, mientras que el método estaría constituido por la implementación de la definición de la función miembro empezarProyeccion(). El mensaje corresponde al *"qué"*, mientras que el método corresponde al *"cómo"*. Así, el método podría definirse -en la clase SalaDeCine- de la forma:

```
void SalaDeCine::empezarProyeccion()
{
        // apaga música y luces y comienza proyección
        ponerMusicaDeFondo( OFF );
        ponerLucesSala( OFF );
        marchaProyector( ON );
}
```

En este caso el mensaje empezarProyeccion podría ser enviado al objeto cineParadox por un objeto cronómetro con un horario determinado.

Advertimos en el código, en lo que parece la definición de una función, una sintaxis extraña: si suprimiéramos la parte "SalaDeCine::" nos quedaría, aparentamente, una porción de código habitual en C. Esta porción "extra" nos sugiere que, de alguna forma que más adelante veremos en detalle, la función void empezarProyeccion(void) "pertenece" o "está circunscrita" a la clase SalaDeCine. El código indica, en definitiva, que estamos definiendo una **función miembro** de la clase SalaDeCine. Hemos visto, también, unas notaciones conocidas para los programadores de C por su uso en "structs": "." y "->". En efecto, las clases en C++ son, en cierta forma, una extensión de los structs (que, ampliados, también existen en C++). Debemos pensar, no obstante, más en la conceptualización como mensajes que como sintaxis de acceso a datos, cual sería más propio de C.

Pudiera parecer, por otra parte, que el método o **función miembro** empezarProyeccion está compuesto por funciones del tipo usado en programación estructurada, pero en realidad se trata de mensajes dirigidos al mismo objeto (en este caso concreto, cineParadox), dado que, por ejemplo, el código

```
marchaProyector( ON );
```

equivale a

```
this->marchaProyector( ON );
```

en donde **this** es un puntero implícito al objeto cineParadox que, como veremos más adelante, "prefija" a todas las funciones miembro no estáticas. Así, y según lo visto antes, el mensaje de poner en marcha el proyector cinematográfico es enviado al objeto por medio de un puntero al mismo. Pero, un momento, un momento: ¿Funciones miembro estáticas? ¿Punteros implícitos? ¿Una función que primero se declara () y más tarde se repite como (void)? De acuerdo, de acuerdo: esto no es fácil. Lo malo de C++ (como de cualquier OOPL) es que muchos de los conceptos expuestos deberán ser retenidos vagamente en primera instancia para poder, luego, con más bagaje, volver a ellos para entenderlos mejor. ¿Significa esto que el lector se convertirá en un re-lector? Pues, efectivamente, sí: en C++ hay que volver a leer muchas veces lo ya leído, pues se parte de una idea básica: sólo se entenderá bien un concepto de C++/OOP si ya se entendía razonablemente bien antes de abordarlo. ¡Pero esto es ridículo!, propondrá el lector. Bien, sólo puedo responder:

"esto es lo que hay". Intenten, si no, explicar a qué sabe el vinagre a quien no lo haya ya probado. Volvamos, pues, a nuestros objetos.

Hemos visto, pues, que el objeto cineParadox responde al mensaje de iniciar la sesión enviándose a sí mismo distintos mensajes de control, que a su vez enviarán otros mensajes (a éste u otros objetos) o manipularán sus propios datos internos. Aparece ahora claro que, en general, el envío de un mensaje a un objeto que carezca del método para responderlo, por sí mismo directamente o mediante herencia o delegación, será calificado, en C++, como un error en tiempo de compilación: lo mismo que ocurriría si llamáramos en C a una función inexistente.

ABSTRACCIÓN

¿En qué consiste la abstracción? Bueno, por decirlo sencillamente, esta es una cualidad de los seres humanos que les permite encontrar lo básico que hay de común en una serie de objetos. Imaginémonos, por ejemplo, a un niño frente a una cesta de fruta: aunque nunca antes hubiera visto esos objetos, en muy poco tiempo podría separar las frutas del mismo tipo. El niño pondría en un montoncito los plátanos, en otro las naranjas, en otro las ciruelas, etc. ¡Pero esto es una bobada!, podría exclamar el lector. ¡Ni mucho menos, señor lector! podría en derecho exclamar entonces yo. La cosa no es nada fácil. Lo que ocurre es que como esta capacidad nos ha acompañado siempre desde niños, no nos damos cuenta cuando la usamos. Es como andar erguido sobre dos piernas: de fácil nada. Y si no que se lo pregunten a los elefantes del circo o a los fisiólogos. Bien. En definitiva por medio de la abstracción conseguimos no pararnos en el detalle concreto de las cosas (si una fruta tiene mayor o menor tamaño, si tiene picaduras de aves, etc.), pudiendo así generalizar y operar con "entes abstractos". O sea, cuando decimos "a mí me gustan las naranjas" no queremos decir que nos gustan "todas" las naranjas, de cualquier tipo y en cualquier estado, sino que indicamos que nos gusta el sabor, la textura, el aroma que por lo general poseen las naranjas. Vemos, de esta manera, que el fenómeno de la **abstracción** consiste en la *generalización conceptual de un determinado conjunto de objetos y de sus atributos y propiedades*, dejando en un segundo término los detalles concretos de cada objeto. ¿Qué se consigue con la abstracción? Bueno, básicamente pasar del plano material (cosas que se tocan) al plano mental (cosas que se piensan). También hemos conseguido volar, construir edificios, la receta de los canelonni Rossini, la bomba de neutrones, etc. Pero ¿qué tiene que ver esto con la OOP? Bien: imaginemos una abstracción sobre un grupo de tornillos. Una vez que descubramos lo que distingue a un tornillo de otros objetos, podremos reconocerlo con cierta facilidad en cualquier sitio, de forma que si aprendemos a manejar un destornillador con un tornillo, lo podremos manejar con cualquier otro, ¡aunque todavía no lo hayamos visto! Pensemos ahora en la OOP: imaginemos que, de alguna forma, hemos podido inocular la capacidad de abstracción a un sistema software. Imaginemos seguidamente que tal sistema reúne las características básicas y funcionamiento de, por ejemplo, varias circuitos lógicos en una estructura (que en C++ se llama *class*). Si codificamos sobre estas características

básicas, en el momento en que queramos introducir un circuito lógico concreto, el trabajo será mínimo o nulo. Bueno, ya iremos viviendo en carne las ventajas de este fenómeno.

ENCAPSULACIÓN

¿Qué es, por otro lado, la encapsulación? Lo que de inmediato nos sugiere el vocablo es la acción de encapsular, de encerrar algo en una cápsula, como los polvitos medicinales que nos venden en las farmacias. Podemos decir que la **encapsulación** se refiere a la *capacidad de agrupar y condensar en un entorno con límites bien-definidos distintos elementos*. El fenómeno de la encapsulación podría darse, por ejemplo, en el conjunto de herramientas y elementos heterogéneos que integran -es un decir- la caja de herramientas de un fontanero: la caja es la cápsula y todo lo que haya dentro es lo que hemos encapsulado. Pero no nos perdamos: la cualidad de "encapsulación" la aplicaremos únicamente a abstracciones: o sea, afirmar que una caja de herramientas concreta *encapsula* un bocadillo, un martillo y un limón constituye una cierta trasgresión léxica. Cuando hablemos de encapsulación en general siempre nos referiremos, pues, a encapsulación abstracta. Las dos propiedades expuestas están, como vemos y en lo que a la OOP concierne, fuertemente ligadas. Dicho de manera informal, primero generalizamos (la abstracción) y luego decimos: la generalización está bien, pero dentro de un cierto orden: hay que poner límites (la encapsulación), y dentro de esos límites vamos a meter, a saco, todo lo relacionado con lo abstraído: no sólo datos, sino también métodos, comportamientos, etc. Pero, bueno ¿y esto es OOP? Pues sí, esta es una característica fundamental de OOP. Veámoslo de esta manera: imaginemos una base de datos relacional conteniendo unos datos accesibles mediante programas en C ó Pascal. Podríamos decir (con cierto recelo) que la estructura de la Base de Datos *encapsula* la información, pero ¿qué podríamos decir de las funciones que se aplican a estos datos? Bueno, lo cierto es que tales funciones o métodos estarán normalmente dispersados en multitud de módulos. En realidad aquí se trata de una *encapsulación a medias* y esto parece que no funciona del todo bien: pensemos en unas cápsulas medicinales que contuvieran únicamente algún componente químico del medicamento y el resto estuviera todo mezclado -en polvo- en el fondo de la caja: quizás al final surtan el mismo efecto, pero, diantre, esto equivaldría a triturar junta toda la comida de un día pensando que, de todas formas, en el estómago se habrá de juntar. Bien: la OOP proclama que una misma *cápsula* contenga datos y funciones. Pero, ¿cómo?, preguntará el inquieto lector. Pensemos en Bases de Datos. Mejor en Bases de Objetos: aquí se archivan a la vez los datos y los métodos para accederlos, manipularlos, etc. ¿Se archivan las funciones? ¡Efectivamente! Podemos comandar un *query* en SQL (mejor en OSQL: SQL de Objetos) que seleccione y ejecute algunas de las funciones archivadas a la vez que se aplica sobre los datos. Al meter en el mismo "saco" (que en C++, como inmediatamente vamos a ver, es la *class*) datos y funciones sí que tenemos una *encapsulación "completa"*.

ABSTRACCIÓN Y ENCAPSULACIÓN EN C++

En lo que a C++ se refiere, la unidad de abstracción y encapsulación está representada por la Clase, (*class*). Por un lado es una abstracción pues, de acuerdo con la definición establecida anteriormente, es en ésta donde se definen las propiedades y atributos genéricos de determinados objetos con características comunes (recordemos el ejemplo de la sala de cine). La Clase es, por otro lado, una encapsulación porque constituye una cápsula o saco que encierra y amalgama de forma clara tanto los datos de que constan los objetos como los procedimientos que permiten manipularlos. Las Clases se constituyen, así, en abstracciones encapsuladas.

En principio, y a efectos de necesaria referencia para el manejo de lo que sigue, adelantaremos una esquemática definición de "*clase*":

*Una **class** en C++ es como un struct del tipo usado en C que admite en su cuerpo tanto variables como funciones, más unas etiquetas que controlan el acceso a ambas*.

Abordemos ahora el tema en su parte cruda y echémosle un vistazo a parte del código de lo que podría ser una descripción de una clase que denominaremos Racional:

```
Class Racional {
friend Racional& operator+( Racional, Racional );
// …
private:
        int numerador;
        int denominador;
        // …
        void simplificaFraccion();
        // …
public:
        // …
        void estableceNumerador( int );
        void estableceDenominador( int );
        // …
};
```

Observemos que en la clase Racional, abstracción del conjunto de los números racionales, de su representación formal (x/y: equis dividido por y) y operaciones (a/b + c/d), aparecen encapsulados tanto los datos internos (numerador, denominador) como las funciones miembro para el tratamiento de éstos (el operador suma de quebrados, el procedimiento para cambiar el denominador, el método para simplificar fracciones, etc.).

Bueno, la verdad es que en el código anterior observamos muchas más cosas: el símbolo de comentario //, que anula lo escrito (a efectos de compilación) desde su aparición hasta el fin de la línea; las etiquetas private y public, que determinan *cómo* y *quién* accederá a los datos y funciones de la clase; la palabra clave friend, que indica la calificación como "amiga de la clase Racional" de una función que suma Racionales (¿Racionales? ¿alguna forma de *typedef*?) y a la que se dará el tratamiento de

"operador suma" (operator +), devolviendo una "referencia" (&) a Racional (¿referencia a Racional? ¿alguna forma de puntero?). Vale, vale: son muchas cosas en las que todavía no podemos entrar y que en su momento veremos en detalle. Bástenos saber, por ahora, que tal código intenta describir el comportamiento y la esencia de un **tipo** de datos bien conocido por todos (el conjunto de los números racionales, de la forma x/y, donde x e y son números enteros) pero no incluido como tipo predefinido[2] por la mayoría de compiladores. Esta carencia es la que intentamos suplir con el código expuesto: una vez descrita la clase (class), se habrá traspasado al compilador el conocimiento de, para él, un nuevo tipo de número: el "Racional". Retengamos de momento únicamente esta última idea pensando, como único alivio, que cuanto más veamos una "rareza", antes dejara ésta de serlo.

Salvado el inciso, quizá puedan apreciarse mejor las características de abstracción y encapsulación notadas en la siguiente porción de la definición de la clase SalaDeCine, tomada del ejemplo "cinematográfico":

```
Class SalaDeCine {
private:
        int aforoSala;
        char *nombreSala;
        // …
public:
        void alarmaEnCasoDeIncendio();
        void empezarSesion();
        // …
};
```

La clase SalaDeCine aglutina, según lo expuesto, tanto los datos correspondientes a las salas de cine en general (capacidad de la sala, nombre de la misma, etc.) como los métodos de respuesta a los mensajes dirigidos a los objetos "salas de cine" (*alarma por incendio*, etc.). O sea, la clase SalaDeCine es la abstracción resultante de la asimilación intelectual de todas las posibles salas de cine, vistas o soñadas: aquí tenemos la cualidad de *abstracción*. La misma clase *encapsula*, por otro lado, datos y funciones dentro de un límite bien-definido: ¿cuál? ¿las paredes del edificio? ¡No, vaya! El límite es el mismo que el que separa una sala de cine de un almacén de frutas: su propia definición, que en este claso coincide con el bloque de la clase. La cápsula esta formada por los brazos {} que encierran el cuerpo de la clase.

Conviene notar que los tipos de datos incorporados al compilador (*int, char, long*, etc.) son realmente abstracciones encapsuladas de datos y métodos. Consideremos el siguiente código:

```
float resta, minuendo, sustraendo;
resta = minuendo - sustraendo;
```

El operador (-) representa aquí la operación de substracción entre dos variables del tipo predefinido 'float'. Pero tal operador es, realmente, un mensaje que el objeto sustraendo (un número *float*) le envía al objeto minuendo (otro número *float*) , y el método de respuesta a tal mensaje está implementado en la encapsulación de tipo **float** predefinida en el compilador, que resta del minuendo el sustraendo y devuelve un valor de tipo float. Tal método es formalmente diferente del que podría ser implementado para la resta de dos variables de tipo int (no hay parte decimal, etc.), o aún de un tipo definido-por-el-usuario (*user-defined*). Aclaremos esto: el compilador sabe que una operación sobre *float*s no tiene por

[2] Cuando hablo de tipos predefinidos o incorporados me refiero a los tipos de datos que el compilador, tal y como sale de su envoltorio original, reconoce: *char, int, long, short, double, float, etc.*

qué ser igual a otra sobre *int*s. Realmente el compilador lo que hace es "colocar" junto a la definición de un tipo de número (en este caso el *float*) las "funciones" específicas para operar con los mismos: prácticamente lo que antes hemos hecho "a mano" mediante las clases (recordemos la clase Racional).

¡Un momento!: demos tiempo a que se asienten las ideas. Primero está el mensaje: hasta ahora parecía que mensaje equivalía a función, a un tipo específico de función (función miembro). Vemos, sin embargo, que aquí la función es sustituida por un operador de substracción. Digamos, por el momento, que un operador es una función con una disposición no ortodoxa de la lista de argumentos. En realidad podríamos considerar que minuendo - sustraendo es equivalente a -(minuendo, sustraendo), y si en esta última expresión sustituyéramos el operador de sustracción por un identificador alfanumérico (del tipo comunmente usado en los lenguajes simbólicos como C), como por ejemplo *restaDeFloats*, tendríamos una función *monda y lironda*. Podemos ver, pues, al operador como a una triquiñuela formalista que nos permite aplicar la notación algebraica clásica a las expresiones de un programa (realmente sólo a algunas de ellas, como ya veremos), aumentando sobremanera la legibilidad del código. Hemos visto también, después, que el objeto a la derecha del operador es el que lanza el mensaje, significado por el operador en sí, al objeto situado a la izquierda de éste. Realmente esto es una norma en C++: el argumento de una función situado más a la derecha en la lista de argumentos lanza el mensaje representado por la función a su elemento contiguo a la izquierda, y así recursivamente. O sea, si tenemos

```
multiplicando1 x multiplicando2 x
multiplicando3 x multiplicando4
```

el esquema de envío de mensajes seguiría el siguiente orden, establecido por los paréntesis:

```
( ( ( multiplicando1 x multiplicando2 ) x
          multiplicando3 ) x multiplicando4 )
```

HERENCIA

Resulta innecesario explicar en qué consiste la herencia biológica (recuerdo que mi padre me espetó una frase similar en mi pubertad). En OOP existe un concepto parejo, en el que los seres vivos son sustituidos por las abstracciones. En C++, en concreto, la herencia se aplica sobre las clases. O sea, de alguna forma las clases pueden tener descendencia, y ésta herederá algunas características de las clases "padres". Si disponemos las clases con un formato de arbol genealógico, tenderemos lo que se denomina una estructura jerarquizada de clases.

Entendámonos: la OOP promueve en gran medida que las relaciones entre objetos se basen en construcciones jerárquicas. Esto es, las clases pueden heredar diferencialmente de otras clases (denominadas "superclases") determinadas características, mientras que, a la vez, pueden definir las suyas propias. Tales clases pasan, así, a denominarse "subclases" de aquéllas. En un esquema simplista basado en grafos de árbol, las clases con las características más generales ocuparían la base de la estructura, mientras que las subclases más especializadas florecerían en los nodos terminales. Veámoslo en un simple -y quizás irreal- ejemplo de jerarquización:

```
Construcciones
        Viviendas
                Edificios
                        Apartamentos
                        Salas de Cine
                        Casas
                        Chalets
        Monumentos
                Contemplativos
                        Estatuas
                                El Pensador, de Rodin
```

Adelantémonos a los acontecimientos: no existe una jerarquía objetivamente correcta. Los mismos 'n' elementos pueden ser ordenados un máximo de 'n!' maneras distintas en jerarquías basadas en distintos criterios, ninguno en principio mejor que otro. Se tiende a pensar, sin embargo, en que el mejor criterio es el más natural, el que se más se acomoda al mundo real. Bueno, esto es sólo una opinión; una muy extendida opinión: lo que se denomina el *criterio antropomórfico*. De momento, empero, obviaremos esta discusión de raíces filosóficas, más propia de las áreas de análisis y diseño. Prosigamos.

En C++ la herencia se implementa mediante un mecanismo que se denomina **derivación de clases**: las superclases pasan a llamarse **clases base**, mientras que las subclases se constituyen en **clases derivadas**.

El mecanismo de herencia está fuertemente entroncado con la reutilización del código en OOP. Una clase derivada posibilita, en C++, el fácil uso de código ya creado en cualquiera de las clases base ya existentes, de la misma manera que uno de nuestros hijos puede heredar alguna de nuestras aptitudes (y la práctica totalidad de nuestros defectos). Consideremos el siguiente ejemplo:

```
class Edificio {
// …
protected:
        int numeroDePlantas;
        char* direccion;
        Fecha fechaConstruccion;
        // …
public:
        virtual void alarmaEnCasoDeIncendio();
        // …
};
```

```
class SalaDeCine : public Edificio {
// …
private:
        int aforoSala;
        // …
        void marchaProyector( boolean );
        // …
public:
        void empezarSesion();
        void alarmaEnCasoDeIncendio();
        // …
};
```

Hemos hecho derivar la clase SalaDeCine de la clase base Edificio (en este caso se ha realizado una *derivación pública*, significada por la notación **:** public Edificio de la cabecera de la clase SalaDeCine, y que explicaremos más adelante). De este modo aprovechamos la abstracción realizada en el nuevo tipo de dato **Edificio**, no teniendo necesidad de volver a implementar en la clase SalaDeCine las variables y métodos ya adscritos a la clase base (efectivamente, la clase derivada puede hacer uso, únicamente restringido por las etiquetas de acceso, de las funciones y los datos de la clase base sin necesidad de redeclararlos). Esto quiere decir, en definitiva, que, aunque no lo hayamos declarado expresamente, la clase SalaDeCine encapsula, también, datos como el numeroDePlantas o la fechaConstruccion. Es como si, al derivar la clase SalaDeCine de la clase Edificio, se añadiera de alguna forma todo lo que hemos declarado en la clase base a la clase derivada. O sea, que al declarar esta derivación lo que estamos haciendo es añadir un pegote de la clase "padre" a nuestra clase "hija". Bueno, la verdad es que físicamente ocurre ni más ni menos que esto (con la excepción de las clases bases virtuales, de las que hablaremos bastante más adelante): la porción del código de la clase base se añade en el módulo objeto a la codificación propia de la clase derivada. Pero, atención, esto no quiere decir que podamos usar indiscriminadamente cualquier dato o función de la clase base. ¡Vaya! ¿Y por qué no? -preguntará el lector-: ¿Para qué queremos añadir un "pegote" que luego no podamos utilizar? Bueno, esto tiene que ver con las "etiquetas" de que hemos hablado antes, y también con el tipo de derivación utilizado. Entonces, ¿hay varios tipos de derivación?. Sí. Existen tres tipos de derivación: *pública, privada y protegida*, como antiguamente decíase que existían tres tipos de descendencia humana: *legítima, secreta y bastarda (o natural)*. Pero, bueno, contestando a la pregunta podría decirse que el bloque del "padre" se traspasa íntegro y luego, dependiendo del tipo de derivación y de las características de las funciones y datos de la clase base, así como de cómo y quién pretenda usarlas, el sistema dirá: "Puede usarse" o "NO puede usarse". Además, es bueno que el padre, o clase base, pueda restringir el acceso a ciertas cosas (piénsese, si no, en el dormitorio conyugal en la vida marital real).

Vemos, no obstante lo anterior, que en ambas clases aparece la misma función miembro alarmaEnCasoDeIncendio(): No se trata de una repetición, sino que, usando de una característica que veremos seguidamente -el ***polimorfismo***-, se ha deseado que un objeto SalaDeCine responda al mensaje de "¡Alarma!, ¡Fuego!" de una forma distinta a como respondería un edificio de uso general (de forma comprensible, pues se da una mayor condensación humana en aquellos locales: esto es, hay que darle a la carne quemada la importancia que merece). Si no existiera esta "duplicidad" un objeto del tipo SalaDeCine respondería al mensaje de alarmaEnCasoDeIncendio con el cuerpo de esta función definido en la clase Edificio, según lo que hemos visto en el párrafo anterior, y quizá no sea esto lo que nos interesa.

El concepto de herencia constituye un estrato básico del paradigma de objetos, pero esto no significa que todas las relaciones entre clases en OOP deban ajustarse siempre a este modelo jerárquico. Es necesario establecer si la pretendida relación entre objetos es de pertenencia o de derivación. En una relación típica de pertenencia un objeto contiene al otro (verbigracia, un objeto coche posee un objeto motor) mientras que en una de derivación un tipo de datos abstracto se extiende constituyendo un subtipo (así se derivaría la clase OsoPanda de la clase Oso). Adelantemos que, según un esquema ampliamente aceptado, las relaciones entre clases de un sistema podrían dividirse en razón de los

tres siguientes predicados y su ajuste a una determinada conjunción: ES-UN, TIENE-UN, ES-COMO-UN. Veámoslo en un ejemplo: un perro *es-un* mamífero, *tiene-una* cola y, para los niños, *es-como-un* juguete. La única relación generadora de un esquema jerárquico es la primera (ES-UN). Esta división, que parece una tontería, es fundamental en la etapa de análisis y diseño, por lo que el lector debería guardársela sin más remora en su mochila de procedimientos. Debemos acostumbrarnos, pues, a que cuando veamos o distingamos dos objetos nos preguntemos: ¿qué tipo de relación los une? Éste es un ejercicio muy recomendado que, puesto en práctica, amortiguará en alguna medida la predisposición que tienen los principiantes en OOP a establecer cualquier relación entre objetos como una jerarquía.

La herencia puede ser simple o múltiple (esta última soportada en C++ a partir de la versión 2.0), dependiendo del número de superclases que se constituyan en base de una dada. La utilidad de la herencia múltiple provocó en principio una fuerte discusión en la comunidad C++, aunque ya parece haberse asentado suficientemente su funcionalidad. La herencia puede establecerse, también y como ya hemos visto, como pública, privada o protegida (esta última a partir de la versión 3.0), dependiendo de la calificación del acceso a lo heredado, como ya veremos detalladamente.

En resumen, el principal beneficio de la herencia en un OOPL, y en concreto en C++, consiste en la fácil reutilización de los componentes de un determinado sistema a través de la captura explícita de la generalización y de la flexibilidad en la incorporación de cambios a la estructura. Recordemos: en OOP pequeños cambios han de generar pequeñas repercusiones. O sea, que podemos aprovechar con poco esfuerzo el código que ya habíamos escrito. Incluso las pruebas de prototipación.

POLIMORFISMO

Esta propiedad, como su mismo nombre sugiere (sólo para los que se manejen en griego: múltiples formas), se refiere a la posibilidad de acceder a un variado rango de funciones distintas a través del mismo interfaz. O sea, que, en la práctica, un mismo identificador puede tener distintas formas (distintos cuerpos de función, distintos comportamientos) dependiendo, en general, del contexto en el que se halle inserto.

En C++ el polimorfismo se establece mediante la *sobrecarga de identificadores y operadores*, la *ligadura dinámica* y las *funciones virtuales*. Vayamos por partes.

El término **sobrecarga** se refiere al uso del mismo identificador u operador en distintos contextos y con distintos significados.

La **sobrecarga de funciones** conduce a que un mismo nombre pueda representar distintas funciones con distinto tipo y número de argumentos. Esto induce un importante añadido de legibilidad al código. Examinemos por ejemplo las siguientes declaraciones, supuestas bien definidas las clases Racional y Real:

```
Real& sumaReal( Real, Real );
Racional& sumaRacional( Racional, Racional );
```

Vemos que funciones con la misma operatividad abstracta -pues en definitiva se rata de sumar números- observan nombres distintos. Y lo malo es que esto ¡nos parece de lo más normal!. En C++ tales funciones podrían ser declaradas así:

```
Real& suma( Real, Real );
Racional& suma( Racional, Racional );
```

de forma que el manejo funcional se torna más intuitivo, facilitando así, a la vez, el mantenimiento del código. Pero, ¿qué ocurre aquí?. Si codificamos lo siguiente:

```
cout << suma( 5, 8 );
```

¿cómo sabrá el sistema cuál de las funciones *suma* tiene que ejecutar? Bueno, en sucesivos capítulos veremos que el sistema aplica a rajatabla unas determinadas reglas en un orden muy preciso, y ayudándose de éstas determina (si puede) a qué función concreta debe llamar.

En el ámbito de la OOP, la sobrecarga de funciones equivale a que un mismo mensaje puede ser enviado a objetos de diferentes clases de forma que cada objeto respondería al mensaje apropiadamente: así el mensaje "suma" dirigido por un objeto Racional a otro obtendría un valor de retorno de tipo Racional (concretamente *referencia a Racional*, un tipo introducido por C++ y que veremos más adelante, y que nosotros muy informalmente podremos, en este momento, asimilar a un puntero constante a Racional con características sintácticas especiales), mientras que el mismo mensaje dirigido a un objeto Real provocaría el retorno de la referencia a un objeto también Real.

Podemos, pues, dotar de más significados al nombre de una función, tanto dentro como fuera de la descripción de una clase. Los programadores de C pueden inmediatamente sopesar la gran ventaja que esto supone: en lugar de tener legiones de funciones conceptualmente idénticas y con distintos identificadores, tales como *sumaDeEnteros, sumaDeFloats, sumaDeDoubles*, etc. (que, por cierto, forman una muralla impenetrable para cualquier lector del código distinto del que lo implementó), la función encargada de sumar números será notada con un único identificador: ***suma*** y, dependiendo del tipo de los argumentos, el trabajo de discernir a qué función concreta se dirige la llamada es dejado al compilador (¡vaya! ¡parece que ya empezamos a ver alguna contraprestación a nuestra inversión!).

La **sobrecarga de operadores** permite, por otro lado, el desarrollo de un código más coherente, como especialización de la sobrecarga de funciones, posibilitando la re-definición (para tipos de datos definidos-por-el-usuario) de las operaciones realizadas por éstos (+, -, *, >, etc.). Esto es, ocurre lo mismo que en la sobrecarga de funciones, pero aquí, en vez de identificadores de funciones, tenemos operadores, que, como ya hemos

comentado, son en definitiva funciones escritas de una forma pretendidamente más clara. De esta manera, el ejemplo anterior podría ser recodificado de la siguiente forma:

```
class Racional {
friend Racional& operator+( Racional, Racional );
// …
};

class Real {
friend Real& operator+( Real, Real );
// …
};
```

pudiéndose usar el nuevo operador como sigue:

```
Real sumaReal, sumandoReal1, sumandoReal2;
sumaReal = sumandoReal1 + sumandoReal2;

Racional sumaRacional, sumandoRacional1, sumandoRacional2;
sumaRacional = sumandoRacional1 + sumandoRacional2;
```

donde aparece evidente que los nuevos tipos definidos-por-el-usuario, que encapsulan las propiedades y atributos de los conjuntos matemáticos de los números racionales y reales, serían tratados por el compilador de igual forma como éste lo haría con los incorporados.

A estas alturas el lector observará asombrado el código anterior y, restregándose los ojos, sin duda concluirá: ¡diantre!. Unas cuantas líneas con una extraña sintaxis y terminología y ¡zas!: sobrecargas, operadores ... ¿qué pasa aquí? Bien, la verdad es que el código expuesto por sí solo no sirve para nada práctico. En las clases, y dado que todavía no hemos llegado al momento de su explicación detallada, únicamente estamos significando "declaraciones" y no "definiciones". Codificamos la declaraciónde una función *suma* u operador '+', pero no decimos nada del cuerpo del método: tenemos la cajetilla de cartón y nos faltan los cigarrillos de dentro. Pero, claro, no podemos atacar a la vez todos los flancos. Lo que se está indicando aquí es que, mediante una propiedad llamada genéricamente *polimorfismo*, se puede codificar de una forma mucho más sencilla y legible, mostrando el aspecto que ofrece en C++.

Es conveniente notar que la sobrecarga de operadores únicamente puede aplicarse dentro del protocolo de descripción de una clase o cuando se toma al menos uno de los argumentos de clase o referencia a una clase. Dicho de otra forma: no podemos cambiar el *modus operandi* de los operadores actuando únicamente sobre tipos pre-definidos. O sea, que si tenemos

```
int entero1, entero2, entero3;
entero3 = entero1 + entero 2;
```

no podemos redefinir aquí el operador '+' para que en vez de sumar enteros, por ejemplo, los reste. Existe también otra restricción en la sobrecarga de operadores: no podemos inventarnos los operadores. Debemos atenernos al conjunto de operadores que ya existe en C++ (que son los de C más algún otro específico como '::' etc.).

Merced a la **ligadura dinámica** -retomando las propiedades polimórficas de OOP- pueden invocarse operaciones en objetos obviando el tipo actual del éstos hasta el momento de la ejecución del código. Los lenguajes compilados normalmente asocian un llamada a una función con una definición particular de ésta en tiempo de compilación, posibilitando así la optimización del código y el chequeo de tipos en la misma. Los lenguajes del tipo de Smalltalk resuelven todas estas llamadas en tiempo de ejecución, proporcionando una gran flexibilidad en detrimento de la eficacia del código. Pensemos, por ejemplo, que deseamos enviar el mensaje "*¡dibújate!*" a un objeto del tipo polígono regular de n-lados y del que ignoramos su subtipo (pentágono, octógono, etc.) en el momento de la compilación (como puede ocurrir en un sistema interactivo de dibujo). Imaginemos que la elección del objeto se realizará por el usuario en tiempo de ejecución. Así, el mismo mensaje dirigido a un objeto triángulo originará una respuesta distinta del recibido si el objeto seleccionado es un hexágono. En Smalltalk, por ejemplo, se pueden definir arrays de elementos de distintos tipos: el compilador no tiene medio de saber el tipo del elemento que será manejado en tiempo de ejecución. ¿De distintos tipos? ¡Entonces no serán arrays! -podría exclamar el lector. Bien, algo de razón hay en esto: dejémoslo en tipo-agregado-*como-un-array*.

C++ provee la ligadura dinámica mediante las denominadas **funciones virtuales** permitiendo, en virtud de un mecanismo íntimamente relacionado con la derivación de clases (y por tanto con la herencia), la redefinición en las clases derivadas de funciones miembro declaradas en las clases base, de forma que será el sistema, en tiempo de ejecución (*run-time*) el que elegirá qué función debe ejecutar, dependiendo del objeto al que vaya dirigido el mensaje. ¡Alto! ¡Alto! ¡Esto no está claro! Veamos: si codificamos un mensaje dirigido a un cierto objeto, ¿dónde está la duda? ¿dónde la capacidad de elección del sistema? Bueno, retrocedamos un poco. Primero debemos notar que existe en C++ una forma de codificar un mensaje de manera que pueda ser dirigido hacia un "conjunto general e indefinido" de objetos, resultando que el objeto al que al final se aplique el mensaje es desconocido para el compilador. ¡Volvemos a lo mismo! -exclamará el lector-: si no está mal entendido, un mensaje equivale, más o menos, a una función, y esta función se aplicará a un objeto concreto: ¡Esto no se entiende! Bueno, vamos a adelantarnos algo en el programa, aunque sea de forma esquemática: Si declaramos una función *virtual* para ser aplicada en un objeto, por ejemplo, de tipo Edificio, lo que está claro es que si el objeto es de otro tipo, por ejemplo *int*, se producirá un error (ya en el momento de la compilación). Sin embargo, si en vez del objeto del tipo Edificio aparece un objeto de una clase *derivada públicamente* de Edificio (como, por ejemplo, SalaDeCine), el sistema buscará una función del mismo nombre y con similar lista de argumentos en la clase derivada y, si existe, ejecutará el cuerpo de ésta en lugar de la que correspondía a la clase Edificio. Hay que tener en cuenta, por otro lado, que estamos hablando de conceptos básicos, por lo que se supone que, ¡demonios!, el lector no tiene por qué entenderlo todo. Sigamos.

Retomemos, una vez más, el ejemplo fílmico mostrando parte de las definiciones de los distintos métodos de respuesta al mensaje alarmaEnCasoDeIncendio dependiendo del objeto al que tal mensaje sea dirigido:

```
Edificio::alarmaEnCasoDeIncendio()
{
        virtual llamarEstacionDeBomberos();
        virtual llamarEstacionDePolicia();
}

SalaDeCine::alarmaEnCasoDeIncendio()
{
        lucesSala( ON );
        marchaProyector( OFF );
        llamarEstacionDeBomberos();
        llamarEstacionDePolicia();
        abrirPuertasEmergencia();
        // ...
}
```

Vemos que dependiendo del objeto a que se dirija el mensaje de alarmaEnCasoDeIncencio la respuesta será distinta. Pero no queda aquí la cosa. Repitámoslo: el esquema operativo de las funciones virtuales es el siguiente: supongamos que poseemos un método o función que efectúa una llamada a la función miembro alarmaEnCasoDeIncencio(). ¿A la función miembro de qué clase? Bueno, aquí esta el quid. Veamos tal llamada como un mensaje dirigido a un cierto objeto, que a la vez es desconocido para el programador al tiempo de la codificación. Imaginemos el mensaje como una bala en la escopeta de un cazador al acecho de cualquier presa de un determinado tipo: cuando aparezca el primer objeto -cuya identidad en principio desconocemos- la escopeta disparará, el mensaje llegará al objeto y éste responderá de una forma **apropiada**: si el objeto es un Edificio con el cuerpo de la primera función del ejemplo; si es una Sala de Cine, con el cuerpo de la segunda. Estamos dejando, de nuevo, que sea el sistema el que se ocupe del trabajo sucio, pues pensemos en la farragosidad de las estructuras de tipo *switch* o *if-else* que deberíamos trabajarnos a mano. A estas alturas ya podemos pensar, pues, en el compilador como en un colaborador y no como en *El Pozo de Babel*.

El lector , llegado a este punto, no debe desesperarse: lo anterior es únicamente un brevísimo acercamiento a varios conceptos clave. Posteriormente habremos de ver en detalle -¡ay!, ¡verdad es que "demasiado" es siempre poco en C++!- todas y cada una de las características notadas. Cabalguemos, pues, sin más, sobre el siguiente capítulo.

3

DONDE C Y C++ DIFIEREN

Hace poco un amigo me comentaba, jocoso, que C++ más bien debiera denominarse '**+C+**': o sea, 'C rodeado de obstáculos, de más y más obstáculos'. Bueno, esto es algo exagerado, pero expresa de forma clara el siguiente esquema: "*Al principio creóse el lenguaje C y se vio que era bueno: se podía disfrutar de él en total libertad. Pero el demonio de la inteligencia tentó al hombre con el árbol de la ciencia: C dando más y más frutos, frutos de más y más. El programador sucumbió a la tentación y, desde entonces, por castigo divino, ya nunca más se daría la flexible sencillez del C de antaño. Fue dicho entonces al desarrollador: cargarás con la doble cruz del trabajo y del chequeo de tipos: la doble cruz de C++, y nunca más trabajarás solo*." Bueno, sólo es una broma (¿o no?). En definitiva quiero indicar que, a ojos del observador alejado, C++ es un C enmarañado y restrictivo. Realmente, por supuesto, esto no es cierto: las restricciones que C++ impone a la flexibilidad de C, sostenidas mayormente en el fuerte chequeo de tipos y en la incorporación de tipos de datos abstractos (clases), suponen un aumento de la efectividad, claridad y mantenibilidad del código, reduciendo grandemente los tiempos de desarrollo y mantenimiento de sistemas software.

Uno de los puntos fuertes de C++ consiste precisamente en su capacidad para ser usado e integrado con facilidad de los sistemas basados en C: la compatibilidad de C++ con C permite, así, aprovechar progresivamente las nuevas características del lenguaje, instando la coexistencia inicial de ambos. Por un lado, y obviando las características de C++ esenciales a la OOP y a sus propiedades asociadas, éste puede ser utilizado simplemente como un *mejor C*, pudiendo de esta manera los sistemas codificados en C hacer uso inmediatamente del fuerte chequeo de tipos de C++, así como de la prototipación y, en general, de las nuevas características de uso general que C++ añade al "sucinto" C. Por otro lado, el conocimiento de las incompatibilidades entre ambos lenguajes nos permitiría evitar en nuestro código C aquellas construcciones generadoras de problemas en la compilación bajo C++. Hay que realizar, pues, en un primer estadio de transición desde ANSI C a C++, necesarios ajustes que permitan la compilación sin problemas de nuestro código C como C++.

En definitiva: en las secciones que siguen veremos, esencialmente, determinadas sintaxis y esquemas de programación comunes en ANSI C, pero incompatibles o generadores de problemas en C++. Si conocemos qué tipo de construcciones C ocasionan problemas al compilar como C++, estaremos en disposición de evitarlas y, a la vez, conseguiremos un mejor código C. Deseo notar que los pequeños ejemplos que se presentan han sido chequeados con Borland C++ 4.0 para la delimitación de los errores, pues éste es, a mi juicio, el compilador para PC's actual que más se acerca a las especificaciones de AT&T 3.0 y del ARM. Veamos, sin más, tales diferencias.

PROTOTIPADO DE FUNCIONES Y CHEQUEO DE TIPOS EN C++

En C++, como en ANSI C, las funciones deben expresamente contener en su declaración el número y tipo de sus argumentos, constituyendo lo que se denomina un *prototipo de función*, pero mientras que una lista de argumentos vacía indica en ANSI C un número indeterminado de argumentos y tipos, la supresión paralela del chequeo de tipos en C++ se consigue mediante la notación de elipsis **(...)**. De esta manera cualquier función no declarada previamente, en primer lugar originará cuando menos un warning, y en el mejor de los casos será asumida por el compilador C++ como del tipo

```
int identificadorDeFuncion( ... );
```

Las siguientes declaraciones en ANSI C

```
extern sumaDeDosEnteros(); // lista indefinida de argumentos
extern void imprimeElMensajeHola( void );          // lista vacía
```

en C++ significan ambas: funciones con una lista de argumentos vacía. Si queremos, sin embargo, expresar en C++ lo mismo que en ANSI C, las expresiones equivalentes serían:

```
extern int sumaDeDosEnteros( ... );      // suprime chequeo de tipo
extern void imprimeElMensajeHola();      // lista de argumentos vacía
```

pues en C++ una lista de argumentos vacía significa "*sin argumentos*". Para asegurar la compatibilidad se aconseja en un primer estadio, aunque en C++ es un anacronismo, el uso de la notación **funcion(void)** para indicar funciones sin argumentos tanto en C como en C++.

De acuerdo con lo anterior, la función main(), en aras de una homogeneidad conceptual, será siempre codificada así:

```
int main( int, char** )          // las funciones siempre deben
{                                                    // declarar sus argumentos
        // aquí viene el cuerpo del código
        return 0;  // la no devolución de valor en una función
                                 // no void causaría un warning en compilación
}
```

NUEVAS PALABRAS CLAVE

C++ se reserva los siguientes identificadores como parte del lenguaje:

asm	*catch*	*class*	*delete*
friend	*inline*	*new*	*operator*
private	*protected*	*public*	*template*
this	*try*	*throw*	*virtual*
volatile	*signed*	*overload*	

El término *overload* ha quedado obsoleto a partir de la versión 2.0 de C++, aunque se mantiene como parte del lenguaje a efectos de la compatibilidad con anteriores versiones.

Por tanto si estamos usando alguna de estas palabras clave en un sistema C, como identificadores de variables o funciones, deberemos renombrarlas para evitar conflictos en tiempo de compilación en el caso de portar nuestro código C a C++.

SALTOS SOBRE DECLARACIONES

Examinemos el siguiente código (si el lector piensa que el ejemplo es demasiado forzado, sólo puedo decir: ¡tiene razón! Aunque yo no me considero tremendista con respecto al uso de la instrucción *goto*, lo cierto es que no se me ocurre una situación que realmente necesite el tipo de salto que describimos aquí):

```
void partidaDeBingoDeSupersticiosos()
{
        numero = extraeBola();
        if ( numero < 1 || numero > 99 )
                goto Fin;                       //Error en C++
        for ( int contador = numero; contador < 99; contador++) {
                switch ( contador ) {
                case 13:
                        cout    << "La empresa es supersticiosa.";
Fin:                    cout    << "\nFin de partida.";
                        return;
                default:
                        cout    << "Ha salido el número: "
                                << numero << "\n";
                        break;
                }
        }
}
```

Vemos que una instrucción **goto** salta condicionalmente al interior del ámbito de un **switch** contenido en el ámbito de un **for** en el que se ha inicializado una variable de tipo int (contador), sintaxis ésta que por otra parte resultará extraña al programador de C y que abordaremos más adelante en el texto. Esto está prohibido en C++, pues podría dar lugar a la desinicialización de una variable no inicializada, al saltar sobre ésta (un poco más adelante explicaremos el "por qué). Una instrucción **goto** podría, no obstante, saltar sobre la totalidad del bloque de ámbito en que se declara la variable. La misma prohibición es aplicable al salto sobre el constructor de una clase desde fuera de su ámbito. ANSI C no contempla esta restricción.

Vemos, a la vez, extrañas construcciones del tipo `cout << X`, que responden a la interpretación en C++ del grupo *printf(...)* de C. Digamos, por el momento, que *cout* es un objeto predefinido para el direccionamiento de objetos al dispositivo de salida estándar, << es el operador de inserción (recordemos aquí lo ya dicho sobre los operadores), mientras que *X* es el objeto a insertar. En definitiva, tal sintaxis resultará en la impresión del objeto X, tal y como, si de manera informal, escribiéramos *printf("%x", X)*. Sigamos.

Básicamente la prohibición de salto notada antes adquiere su verdadero sentido en el salto sobre un ámbito en el que se han inicializado constructores de clases. La llamada, explícita o no, un destructor de un objeto cuya inicialización ha sido "saltada" en el mismo ámbito podría ocasionar problemas. ¿Constructores? ¿destructores? ¿algún tipo de Terminators? Ok: está bien: todavía no hemos hablado de esto. Adelantaré que un constructor es un código que suple en las clases, en su calidad de tipos definidos-por-

-el-usuario, los procedimientos de inicialización con que el compilador provee a los tipos predefinidos. Pensemos en un array de *char*s: el compilador reserva espacio en memoria, asigna un puntero a la primera celda, etc. Pensemos ahora en un array de *Letra*s, un tipo nuevo (para el compilador, claro) descrito en una clase Letra codificada por nosotros: se necesitan realizar parejas maniobras para su inicialización: aquí entran los constructores. Es fácil imaginarse que función tienen los destructores. ¡Pero entonces, ¿Cómo ...? Vale, vale: éste es un asunto serio: lo aplazaremos hasta verlo en detalle.

Podemos apreciar el problema en el siguiente ejemplo, en el que un **goto** salta por encima de la inicialización de un objeto de tipo definido-por-el-usuario (Racional), produciéndose una catástrofe cuando se destruye, al final del ámbito, un objeto que no ha sido inicializado.

```
void controlaCortoRecorrido( int espacio, int tiempo )
{
        if ( !espacio )
                goto SinRecorrido;              // ERROR en C++
        if ( tiempo ) {
                // en la línea siguiente se inicializa
                //  un nuevo objeto del tipo Racional
                Racional* velocidad = new Racional(espacio,tiempo);
                cout << "La Velocidad es: " << velocidad;
SinRecorrido:   cout << "\n";
                // Seguidamente, como fin del ámbito de la
                //  estructura de control if, entraría en acción
                //  el destructor del objeto "velocidad",
                //  desinicializándolo.
        }
}
```

Realmente es muy probable que el lector no haya comprendido del todo en qué se basa esta "prohibición del salto". Pero lo cierto es que no puedo explicarlo -por el momento- de un modo mínimamente inteligible sin extenderme bastante en otros conceptos en los que todavía no hemos entrado y que, en su día, llevarán bastante trabajo para su asimilación. El lector deberá tomar esta prohibición, pues, como una orden: "NO USE *GOTO* para este tipo de salto". Y yo incluso la simplificaría de la siguiente forma: "EN NINGÚN CASO USE *GOTO*". A lo largo de mi vida profesional he confeccionado, leído, revisado y estudiado miles y miles de líneas de código C++ comercial, y puedo afirmar que **jamás** he encontrado una simple instrucción *goto* en ellas. Si en C el uso de *goto* es discutible, en C++ simplemente es innecesario y la mayor parte de las veces perjudicial (¡vaya! -dirán ustedes- y eso que decía no ser maniático con respecto a este tema). Bueno, esto es como lo del sexo: empiezas con casi nada y enseguida te apasionas.

INICIALIZACIÓN DE ARRAYS

Si en la inicialización de un array se indica su tamaño, en C++ éste debe corresponder exactamente al del inicializador, como vemos en el siguiente ejemplo autoexplicativo:

```
// La siguiente línea causa error en C++, pero es OK en ANSI C
char* aviso[44]="No se ha tenido en cuenta el byte nulo final"
// la siguiente línea es OK en C++
char* aviso[45]="Sí se ha tenido en cuenta el byte nulo final"
```

Dado, por otro lado, que la siguiente definición en ANSI C

```
char* holaMundo[4] = "hola";                    // Error en C++
```

es equivalente a

```
char* holaMundo[] = "hola";                     //OK en C++
```

se recomienda, pues, adoptar esta última forma para evitar la incompatibilidad del código, pues con esta última sintaxis no tendremos que 'contar' y evitaremos posibles errores.

CONVERSIÓN POR ASIGNACIÓN DE PUNTEROS A VOID

En ANSI C la conversión de un puntero de tipo **void*** a cualquier otro tipo se puede realizar de forma implícita mediante una asignación, mientras que en C++ esta conversión, como posible generadora de errores muy difíciles de depurar, requiere un **cast** explícito.

Los punteros a *void* se suelen declarar en la construcción de componentes genéricos de software, pues pueden ser asignados como punteros a cualquier otro tipo (incorporado o definido-por-el-usuario).

Examinemos el siguiente código, que compilará correctamente en ANSI C:

```
// ...
char* dato = "hola"
void* punteroADato = &dato
int* punteroAEntero = punteroADato;   // error en C++
// ...
int resultado = 100 * (*punteroAEntero)          // INDEFINIDO!
// ...
```

Hemos convertido, implícitamente, por medio de la asignación a punteroAEntero, un puntero a *char* en un puntero a *int*, lo que puede producir un resultado erróneo o, simplemente, inesperado (el anterior código compilado con Borland C++ 4.0 en mi PC asigna a resultado el valor 8400). Este código no compilaría en C++, que exigiría un **cast** expreso, como por ejemplo

```
int* punteroAEntero = (int*) punteroADato;
```

asegurando así la intención del desarrollador en la conversión. O sea, que forzando al programador a expresar mediante una codificación expresa su deseo de realizar esta conversión, el sistema ya puede "lavarse las manos".

La compatibilidad entre C++ y ANSI C impone, pues, la conversión expresa (*cast*) del puntero a void* antes de la asignación a un puntero a otro tipo.

Un problema típico derivado de esta restricción sería el ocasionado por el uso del identificador NULL, definido en muchos archivos de cabecera para uso de ANSI C de la forma

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

así que una asignación de NULL a un puntero a cualquier tipo, como por ejemplo

```
// error en C++
int* punteroAInt = NULL;//equivale a punteroAInt = (void*)0
```

sería flagelada en C++ como error en tiempo de compilación. Una solución a este problema pasaría por la inclusión en el archivo de cabecera del siguiente código:[3]

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

Aquí hemos usado de una macro del preprocesador (*__cplusplus*) que veremos un poco más adelante y que, básicamente, nos permite saber si el código está siendo compilado como C++ o no.

[3] Incidentalmente, y como corolario del espinoso tema de la definición de `NULL`, debemos notar que la portabilidad del código podría verse truncada, también, por declaraciones como la siguiente, encontrada en ficheros de cabecera DOS para el modelo de memoria large:

```
#define NULL 0L
```

pues 0L no puede convertirse en un puntero. Afortunadamente un grupo cada vez mayor de vendedores de software evita la inclusión directa de código no-portable en sus productos.

TIPO DE ENLACE EN VARIABLES CONSTANTES GLOBALES

Las variables globales -declaradas fuera del ámbito de cualquier función- con tipo **const** se consideran en ANSI C con tipo de enlace **extern**, mientras que en C++ se enlazarían como **static**.

Esto quiere decir que si compilamos como C++ un módulo codificado en ANSI C conteniendo variables globales constantes, y en otros módulos se producen referencias a estas variables, se producirá un error en el enlace.

La compatibilidad entre ANSI C y C+ a este respecto puede conseguirse declarando expresamente como **extern** tales variables de tipo **const**:

```
const Racional mitad( 1, 2 );// enlace static por defecto en C++
extern const Racional tercio( 1, 3 );// fuerza enlace extern
```

Si seguimos esta indicación, el código en ANSI C podría ser portado sin problemas a C++.

ENLACE DE TIPO SEGURO

Resulta que las funciones, funciones-miembro, clases y datos-miembro en C++ no se codifican en los modulos objeto con la misma simplicidad que en C. ¡Vaya! Pero, ¿por qué? Cuando antes hemos visitado brevemente el *polimorfismo*, hemos visto que, por ejemplo, funciones con el mismo nombre pueden tener cuerpos distintos. Basándonos en el contexto del código, el lector podría fácilmente adivinar qué función debe ser ejecutada (dependiendo sobre todo de los argumentos). Sin embargo, si estas funciones fueran codificadas 'sin más' en los módulos objeto, tal y como se hace en C, tendríamos un montón de identificadores repetidos, sin procedimiento que nos permita elegir adecuadamente. Aquí entra en acción la codificación interna, y lo que ésta hace es, básicamente, añadir las características del contexto (argumentos, nombre de la clase, etc.) al nombre de la función, o añadir algún símbolo identificativo a ciertas estructuras especiales como clases y variables como datos-miembro.

Basada en el trabajo del Dr. Stroustrup "*Enlace de tipo-seguro en C++*", la codificación interna (*name mangling*) de las funciones en C++ en los módulos-objeto es realizada añadiendo al nombre del identificador de la función los datos relativos al tipo de sus argumentos y, si es el caso, la clase a la que pertenecen como miembros. En versiones anteriores a la 2.0 se producía una deformación únicamente en la codificación de las funciones antecedidas por la palabra clase **overload**, basada en el orden de sobrecarga de éstas y a fin de evitar la ambigüedad en las llamadas a las mismas. A partir de la versión 2.0, y debido a los fallos de eficacia observados a este respecto en versiones anteriores, la codificación "*enlace de tipo-seguro (type-safe linkage)*" se produce siempre para todos los identifi-

cadores de función, resultando en una verdadera comprobación de tipos por parte del enlazador, por lo que la clave **overload** ha quedado obsoleta.

Teniendo en cuenta esta característica y si compilamos como C++, toda función C incluida en un programa C++ también sería codificada con el esquema *enlace de tipo-seguro*, lo que originaría un error al intentar enlazarla con su librería correspondiente, pues el enlazador esperará encontrar en las librerías la codificación especial de la función, y en éstas solamente encontraría la codificación interna normal en C. Para evitar este problema, la supresión de tal codificación interna debe notificarse expresamente al compilador mediante una directiva de enlace de la forma

```
extern "LENGUAJE" { /* Codificación */ }
```

donde "LENGUAJE" habrá de ser sustituido por el código correspondiente ("C++", "C", "Pascal", "Ada", etc.)[4].

Así, por ejemplo, y como norma general en las librerías y compiladores comerciales, las funciones de tipo C son codificadas como sigue en sus respectivos archivos de cabecera:

```
extern "C" size_t strlen( char* );          // string.h
extern "C" char* itoa( int, int );          // stdlib.h
extern "C" char* gets( char* );             // stdio.h
// ...
```

o simplemente

```
extern "C" {
extern size_t strlen( char* );
// ...
}
```

o aún

```
extern "C" {
#       include <stdio.h>
// ...
}
```

Pero esta codificación con la sentencia clave *extern "C"* impediría la compilación en C, por lo que, para asegurar la compatibilidad, deben ser usadas las macros del preprocesador:

[4] Hay que notar que el lenguaje C++ únicamente asegura la validez de los códigos "C++" y "C", siendo los restantes ("FORTRAN", por ejemplo) dependientes de la implementación específica del compilador.

```
#ifdef __cplusplus
        extern "C" {
#endif
extern size_t strlen( char* );
extern "C++" peticionSaldo();          /* agujero C++ selectivo en la co-
                                              dificación de funciones como C */
// …
#ifdef __cplusplus
        }
#endif
```

No se permite, por otra parte, el uso de tal directiva de enlace dentro del cuerpo de definición de una función, así como su uso de ninguna manera implica la conversión de tipos entre lenguajes de programación (strings en FORTRAN a strings en C++, etc.), siendo responsable del explícito ajuste el programador. O sea, que esta directiva, como ya ha sido dicho, únicamente afecta al *name mangling* de los identificadores. Nada más. Y nada menos. Pensar que una simple clave podría "transformar" código de un lenguaje a otro, sin más, es pensar demasiado. Bajemos a planeta Tierra (y digamos, de paso, que existen herramientas comerciales para llevar a cabo esta tarea, aunque éste no es nuestro tema).

Es interesante notar que la declaración extern "C" aplicada a la definición de una función contenida en un módulo C++ funcionará correctamente, causando que tal función (no sobrecargada, en previsión de pausibles errores de compilación) no sea internamente codificada por el esquema de *enlace de tipo-seguro*. Tendríamos, así, una función compilada en C++ con tipo de enlace C. Esta técnica puede ser empleada, por ejemplo, para la llamada desde C a funciones miembro de clases en C++.

Retomando el esquema de enlace y de acuerdo con lo anterior, la función

```
int sumaDeEnteros( int, int );
```

sería codificada internamente en C++, por ejemplo, como **sumaDeEnteros__Fii**, según el método propuesto por el Dr. Stroustrup, donde *F* indica *función* e *ii* corresponde a los dos argumentos de tipo int (*i*). De cualquier forma, la codificación interna de nombres depende de la implementación C++.

En el caso de Borland, tal y como aparece en su *"Manual de Arquitectura Abierta"*, tal codificación interna sigue, básicamente y en una forma resumida poco rigurosa, el siguiente esquema:

```
[@NombreClase][@nombreFuncion$qArgumentos || @datoMiembro]
```

De esta forma, y con la ayuda de códigos auxiliares, la misma función anterior sería codificada internamente por Borland C++ como **@sumaDeEnteros$qii**. Como podemos apreciar en ambos ejemplos, el tipo de retorno de la función en ningún caso es codificado, lo que ya nos anticipa que este factor no decidirá en la resolución de sobrecarga de funciones.

El uso de depuradores (*debuggers*) sin mecanismos para decodificar esta encriptación de nombres ha causado no pocos quebraderos de cabeza a muchísimos programadores de C++. Se aconseja, pues, repasar concienzudamente la documentación de su implementación C++.

ÁMBITO DE *STRUCT*'S

En ANSI C el bloque **struct** no posee ámbito propio, sino que éste es traspasado al ámbito del bloque en el que anida. De esta manera, por ejemplo, el siguiente fragmento es ilegal en ANSI C:

```
struct estadoImpresora {
        enum { APAGADA, ENCENDIDA, ESPERA } status;
        // ...
}
char* ESPERA = "Impresora en espera";
```

arrojando un error por reutilización del identificador ESPERA en el mismo ámbito.

En C++, sin embargo, **struct** define su propio ámbito, por lo que las enumeraciones declaradas en su bloque son locales a éste. El código anterior es legal, pues, en C++. O sea que, ojo a lo que contienen los *struct*s en nuestro código C. Una buena forma de conocer los problemas es intentar compilar como C++ lo codificado en C, e ir revisando los errores y warnings uno a uno.

CONSTANTES LITERALES DE CARACTERES

En ANSI C una constante literal de carácter posee tipo **int**. Esto es,

```
char letraPorno = 'X';
sizeof( letraPorno ) = sizeof( int );       // Ok en ANSI C
```

En C++, sin embargo, una constante literal de carácter posee tipo **char**:

```
sizeof( letraPorno ) = sizeof( char );
```

de forma que en C++ no es necesariamente cierto[5] que

[5] Cuando decimos que algo *no es necesariamente cierto* en C++, esto significa que ese "algo" en cuestión depende de la implementación C++. Es decir, en un sistema dado un *char* puede ocupar los mismos bits que un *int*, mientras que en otro estos dos tipos pueden tener tamaños distintos. La estandarización de C++ únicamente impone que un determinado tipo debe tener un tamaño igual o mayor

```
sizeof( char ) = sizeof( int );          // Ok en ANSI C
```

Esta característica de C++ ha sido implementada para evitar ambigüedades en la sobrecarga de funciones, como veremos más adelante.

TIPO DE LOS ENUMERADORES

Un enumerador posee tipo **int** en ANSI C, mientras que posee el tipo de su enumeración en C++. Esto es, en el siguiente código

```
enum boolean { FALSO, VERDAD };
boolean soyGuapo, tengoDinero;
```

las variables soyGuapo y tengoDinero serían de tipo **int** en ANSI C, mientras que en C++ serían de tipo **boolean**.

De esta forma la expresión

```
sizeof( soyGuapo ) = sizeof ( int );
```

que es siempre cierta en ANSI C, no lo es necesariamente en C++.

MACRO __cplusplus

Si se desea construir un código que compile indistintamente en C y C++, mezclando ambos estilos y declaraciones, debe usarse la macro **__cplusplus** junto con las directivas condicionales del preprocesador **#ifdef** e **#ifndef**. Por ejemplo

```
#ifdef __cplusplus
        const limiteVector = 10;
#endif    /* __cplusplus */
#ifndef __cplusplus
        #define limiteVector 10
#endif    /* __cplusplus */
```

Se pueden aprovechar las características de macro-expansión de las directivas del preprocesador para delimitar perfectamente las particularidades de C++ y C, de forma parecida a como se mostró en la descripción del *enlace tipo-seguro* expuesta anteriormente.

---

que otro, pero no impone condiciones y deja libertad en este sentido a la implementación de C++.

De esta manera, si el código se está compilando como C++, *_cplusplus* se asimilará como definido y se compilará, en el ejemplo anterior la línea que empieza con *const*. En otro caso, se compilará la otra sentencia.

DEFINICIONES MÚLTIPLES

En tanto que en C++ una declaración de variable sin el cualificador **extern** será asumida como una definición -dejando al desarrollador la responsabilidad de diferenciar entre declaración y definición-, en ANSI C se permiten las declaraciones múltiples *de variables globales*, considerándolas "*definiciones tentativas*" y promoviendo a definición únicamente a una de ellas. Por ejemplo:

```
char* cineParadox;
char* cineParadox; //error en C++: múltiple inicialización
// …
int main( int, char** )
{
        char* cinePalafox;
        char* cinePalafox;  // error en ANSI C y en C++
        // …
        return 0;
}
```

Las declaraciones de variables sin el cualificador **extern** son consideradas como definiciones en C++ debido fundamentalmente a una cuestión de compatibilidad y homogeinización conceptual de los tipos incorporados con respecto a los tipos definidos-por-el-usuario (*clases*). Implementemos, por ejemplo, una clase que encapsule las características del tipo de dato *float*:

```
class Float {
private:
        float numero;
// …
};
```

La sentencia

```
Float numeroConComaFlotante;
```

corresponde a una definición en ambos, C y C++, mientras que

```
float numeroConComaFlotante;
```

es considerada como una declaración en ANSI C y como una definición en C++. Esta inconsistencia conceptual es la que se ha pretendido evitar en C++, por lo que una declaración en este lenguaje de, por ejemplo, un *int* habría de realizarse de la forma:

```
extern int declaracionDeInt;
```

El programador de C no acostumbra a separar de forma clara las definiciones de las declaraciones, por lo que lo anterior le puede parece, al menos, confuso. En C++ pueden existir múltiples declaraciones, pero sólo una definición. Ante la duda de si una expresión constituye una definición o una declaración, lo más expeditivo es aplicar la palabra clave *extern*. En el "Libro de Respuestas" de Tony Hansen pueden encontrarse multitud de ejemplos y ejercicios sobre éste y otros temas.

ESPACIO ÚNICO DE NOMBRES DE IDENTIFICADORES Y ESTRUCTURAS

Las estructuras *class*, *struct* y *union* comparten en C++ el espacio de nombre con el resto de los identificadores, de forma que el nombre de una estructura local puede ocultar la visibilidad de un identificador de ámbito exterior a la estructura. Por ejemplo,

```
int direccion = 1;
void introduceDatosClientes() {
        struct direccion {
                char* tipoViaPublica;
                char* calle;
                int numeroDePolicia;
                char* codigoPostal;
                char* ciudad;
                char* provincia;
        };
        // la siguiente sentencia imprimirá el número 1
        //  en el dispositivo de salida estándar
        //  si este código se compila en ANSI C,
        //  procurando, sin embargo, un error en C++
        //  por uso impropio del identificador
        printf( "%i\n", direccion + 0 );          //error en C++
        // la siguiente sentencia equivale, en ANSI C, a
        //  sizeof( int )
        printf( "%i", sizeof( direccion ) );
// ....
}
```

La compatibilidad puede salvarse con el uso explícito antes del nombre de uno de los cualificadores **enum**, **class**, **struct**, **union** o **::**. De esta manera las últimas líneas del código anterior podrían ser reescritas de la siguiente forma:

```
cout << ::direccion + 0;                    // Ok en C++
cout << sizeof( struct direccion );
```

Aquí hemos usado el operador cualificador de ámbito (**::**) para adscribir el identificador usado al correspondiente a la variable global, como veremos en sucesivos capítulos.

Esta restricción ha sido impuesta en C++ para permitir el uso de la sintaxis para constructores de clases:

```
class SalaDeCine {
// …
public:
        SalaDeCine() {};
        SalaDeCine( char* );
// …
};
SalaDeCine cineParadox = SalaDeCine( "Cine Paradox" );
```

4

DONDE C++ SE DESTAPA

¿Dónde nos habíamos quedado? Bien, en el capítulo anterior pudimos repasar brevemente los cambios que han de aplicarse a nuestro código ANSI C para que pueda compilar sin mayores problemas como C++. O sea, el primer paso en un camino que oportunamente iremos sembrando de una suerte de migas de pan (los "objetos" de Pulgarcito), de forma que podamos en cualquier momento desandar lo andado y mirar de nuevo el mapa que nos conducirá al final de esta serie. No nos perdamos, pues. Bien, en esta ocasión vamos a atacar las características de C++ que suponen una mejora sobre C, pero sin entrar todavía en el nucleo del nuevo lenguaje: *la clase*. Esto es, vamos a fijar nuestra atención en los posibilidades de C++ no entroncadas directamente con la OOP, de forma que podrían ser incluidas incrementalmente en nuestro código C. Estas características, que suponen mejoras efectivas sobre C, resultan en un aumento real de la eficacia de este lenguaje, a la vez que proporcionan un soporte más adecuado para el desarrollo de aplicaciones. Pero antes de seguir, y como ya es costumbre, una nota al lector avisado: rigor, brevedad, C++, introducción y claridad son asuntos de difícil conjunción (que no imposible), así que ¡paciencia!. El apartado dedicado a los operadores **new** y **delete**, por ejemplo, reúne y explica algunos aspectos difíciles de encontrar juntos en otras fuentes, con la posible desventaja de tener que involucrar conceptos sobre *clases* que sólo se comprenderán efectivamente más adelante. Pero, bueno, ya sabemos que en C++ debemos acostumbrarnos a oscilar adelante y atrás, más y más, en nuestras lecturas y en nuestro código. La parte dedicada a la sobrecarga de funciones, sin embargo, no podrá ser todo lo extensa que yo quisiera: aquí no se trata de confeccionar un manual de referencia ni una descripción completa del lenguaje, de forma que no podemos pararnos en "el planteamiento del problema del análisis preparatorio del C++", parafraseando a Heidegger. Un poco de aquí y un poco de allí: algo sobre lo que el lector pueda basarse en su primer acercamiento al lenguaje. Empecemos sin más dilación.

DELIMITADORES DE COMENTARIO

Bueno, esto ha llegado a convertirse en una suerte de norma: cuando se crea un nuevo lenguaje parece casi obligada la creación de unos particulares delimitadores de comentarios ('%' para Turbo Prolog, '*' en Fortran, 'rem' en el antiguo Basic, etc.), como si el propio lenguaje no fuera ya suficientemente diferente. Bien: C++ no podía ser menos. En principio, y en aras de esa siempre buscada compatibilidad, C++ admite los delimitadores de comentario usados en C, de forma que, como en este lenguaje, el preprocesador *sustituirá por un espacio* el bloque empezando en '/*' hasta encontrar '*/'. Por ejemplo:

```
/*
        Los delimitadores de este comentario
        corresponden al viejo estilo C.
*/
```

Estos delimitadores, como es bien sabido, son usados para encerrar varias líneas de código y no se pueden anidar: esto es, el código

```
f = m * a /* m /* masa */ * a /* aceleración */ */;
```

resultará en el siguiente error

```
f = m * a * a */;
```

Aún más frecuentemente se da el error de la siguiente forma, normalmente producido al desechar parte del código ya escrito con sus propios comentarios:

```
/*
void solicitaClaveDePaso()
{
        cout << "Introduzca clave de acceso: ";
        cin  >> clave;          /* mi clave: LOGOS */
        validaClave( clave );
}
*/
```

En este caso resultaría el error

```
validaClave( clave ); } */
```

Algunos compiladores comerciales permiten el anidamiento de comentarios, lo que permitiría preprocesar sin problemas el anterior ejemplo, pero esta práctica debe evitarse en aras de la portabilidad del código.

Seguidamente, como ya habíamos avisado y el lector ha podido apreciar en los capítulos precedentes, una vez aceptado el viejo estilo de comentarios en C, C++ introduce un nuevo delimitador: '**//**', que fuerza al compilador a

ignorar todo cuanto se halle incluido desde el mismo delimitador hasta el final de la línea.

Debe notarse que aunque en C y C++ los espacios no suelen ser significantes, si se introduce uno o más entre los dos caracteres de cualquiera de los delimitadores notados, éstos dejarán de constituirse en tales: o sea, los delimitadores de comentarios en C++ son grupos de dos caracteres unidos sin espacios. De acuerdo con esto, las siguientes líneas, por ejemplo, procurarían sendos mensajes de error:

```
f = m * a;                    / * fórmula de la fuerza * /
v = e / t;            / / fórmula de la velocidad
```

Existen, por otro lado, situaciones curiosas, señaladas en la mayoría de los manuales elementales sobre C++, como la siguiente:

```
double v = e //* dividido por tiempo */ t;
;
```

donde si compilamos como C++ resulta

```
double v = e;
```

mientras que al compilar como C aparece

```
double v = e/t;
```

Este problema puede evitarse insertando un espacio tras el operador '**/**' así:

```
double v = e / /* dividido por tiempo */ t;
```

En general un comentario puede insertarse en cualquier lugar del código dondequiera que también sean legales un espacio, una tabulación o un código de final de línea.

Por último, una advertencia: debe evitarse el uso de comentarios de la forma '//' en las macros del preprocesador, pues la expansión de éstas podría provocar "problemas". El siguiente código, por ejemplo,

```
#define HABITACIONES 7   // ¡vaya palacio!
// ...
int superficieHabitable[ HABITACIONES ];
```

producirá tras la macro-expansión la línea

```
int superficieHabitable[ 7   // ¡vaya palacio! ];
```

resultando en el código erróneo

```
int superficieHabitable[ 7
```

Esto puede evitarse usando el viejo estilo de comentarios de C con las directivas del preprocesador, como por ejemplo

```
#define 2PI 6,28306          /* doble de PI */
```

Bien, queda por considerar la cuestión: ¿cuándo se debe usar uno u otro delimitador? En teoría el estilo C debería utilizarse para encerrar bloques de varias líneas, mientras que los nuevos delimitadores parecen más apropiados para comentarios puntuales de líneas concretas de código. ¿Qué ocurre, sin embargo, en el planeta Tierra? -como diría Allen-. Lo cierto es que si examinamos un archivo de cabecera típico de C++ encontraremos una letanía interminable de '//' situadas al inicio de cada línea, donde pudiera parecer mucho más apropiado y requeriría muchas menos pulsaciones el viejo estilo C: o sea, en el fondo a los desarrolladores de C++, como comentábamos al principio, nos gusta diferenciarnos y, ¡demonios!, ¿por qué no? Para el principiante puede significar también un aliciente pensar que, al comentar de esta nueva forma su primera línea de código, ya está "desarrollando en C++". ¡Bendita salvedad!

OPERADOR CUALIFICADOR DE ÁMBITO (*::*)

El operador '**::**' posee dos usos en C++. Por un lado puede utilizarse para acceder a una variable global cuya visibilidad ha sido ocultada por una variable local. Veámoslo en el siguiente ejemplo:

```
// ...
// la siguiente función fue definida anteriormente
int edad = calculaEdad( fechaNacimiento);
const int edadMinima = 18;    // siempre mejor que usar #define
void intentaFranquearAccesoABar( int edad )
{
        //  sólo modificará la variable local edad
        if ( ::edad < edadMinima ) // chequea la variable global
                edad = 16;          // cambia el valor de la variable local
        cout    << "Déjeme pasar al bar: ya tengo "
                << edad << " años.";
        // ::edad no ha cambiado en ningún caso
}
```

Bueno, el ejemplo está un poco "traído por los pelos", pero nos permite recordar que debe tenerse en cuenta que el ámbito del parámetro formal de una función se limita al ámbito local delimitado por el bloque constituido por ésta. La notación **::edad** se refiere a la variable global edad de tipo constante *int*, mientras que la variable local *edad* contenida en el ámbito local de la función oculta la visibilidad de aquélla.

En el ejemplo que sigue, bastante más didáctico, podemos observar cómo el cualificador de ámbito sólo permite el acceso a la variable global, imposibili-

tando el acceso a variables con la visibilidad solapada pertenecientes a ámbitos intermedios. Veámoslo en la práctica:

```
float i = 13.13;                    // variable global
void limitacionAccesoAVariable()
{
        char i = '?';               // inaccesible desde el siguiente 'for'
        for ( ;; ) {                // 'forever'
                int i = 0;  // variable local al bucle 'for'
                // desde este bloque local no se puede acceder a la
                //  variable local 'i' de tipo 'char'
                //  del bloque local intermedio.
                cout    << "Variable local de tipo int: "
                        << i++ << "\n";
                cout    <<"Variable global de tipo float"
                        << ::i << "\n\n";
                if ( i > limiteIteraciones )
                        break;
        }
}
```

Y esto no cambiaría por el hecho de que en este caso no existiera variable global. O sea, el operador '::', por expresarlo gráficamente, practica un agujero en el bloque o estructura local a través del cual "trae" a éste una variable global: nada más (y nada menos). Esto nos proporciona un nivel adicional de flexibilidad en la no siempre fácil elección de los identificadores.

El operador '**::**' se usa también en C++ cuando una función miembro de una clase se define fuera del bloque en que ésta se declaró. Bueno, ya salió: ni habiéndolo anunciado en el preámbulo podemos libranos de las clases: al fin y al cabo estamos en C++. Intentaré explicarlo mediante un ejemplo:

```
class Racional {
private:  // etiqueta cualificadora de acceso privado
        int numerador;                  // dato miembro de la clase
// …
public:   // etiqueta cualificadora de acceso público
        // sigue una declaración de una función miembro
        void estableceNumerador( int );
// …
};

void Racional::estableceNumerador( int numero )
{       // definición función miembro
        numerador = numero;
}
```

Lo que en realidad ocurre es que el operador '**::**' *cualifica* a las funciones miembro a las que se aplica, permitiéndoles el acceso al protocolo de descripción de sus respectivas clases. De hecho, y siguiendo con la sintaxis "gráfica" expuesta poco antes, podríamos decir que la construcción con sintaxis '*nombreDeClase::*' produce un "agujero" a través del que se trae una

porción del ámbito de la descripción de la clase a la función que estamos definiendo, de forma que en realidad la estamos definiendo "virtualmente dentro" de la clase.

Si se ha entendido bien el funcionamiento del operador, el lector podrá ahora apreciar que lo que al principio se han establecido como dos usos distintos de éste, se corresponden a una misma acción, pero con distinta sintaxis. Usando ahora un símil parejo al expuesto anteriormente, podríamos decir que el operador '::' coloca lo que tiene a su derecha en el ámbito de lo que aparece a su izquierda (una clase o "nada" en el caso del ámbito global).

DECLARACIONES MEZCLADAS CON EXPRESIONES

¿Quién no ha observado en un sistema codificado en C una enorme acumulación de declaraciones de variables situada bien al principio del código bien inmediatamente de los bloques en que éstas serían definidas? ¿No se constituye, por otro lado, en motivo de orgullo de muchos programadores en C -y en otros lenguajes "estructurados"- entre declaraciones y definiciones de variables? ¿Quién no se ha perdido alguna vez en el largo y complejo código de una función intentando seguir la pista a identificadores de señales e iteraciones?

Una codificación típica de C podría ser la siguiente:

```
long fila, columna;
/* algunas líneas de código */
for ( fila = 0; fila < maximoFilas; fila++ )
        for ( columna = 0; columna < maximoColumnas; columna++ )
                cout << matriz[ fila ][ columna ];
```

En C++, empero, se relaja al límite la obligación que C impone sobre la precedencia de las declaraciones sobre variables antes de su uso en el cuerpo de una función o en el ámbito global de un módulo. De esta forma C++ permite la mezcla sin restricciones de las declaraciones con las expresiones, resultando que el anterior código podría ser reescrito en C++ así:

```
for ( long fila = 1; fila < maximoFilas; fila++ )
        for (   long columna = 0;
                        columna < maximoColumnas; columna++ )
                cout << matriz[ fila ][ columna ];
```

Hay que notar que el ámbito de una variable declarada dentro de una expresión no se limita al bloque que ésta define, sino que se extiende desde el lugar de declaración hasta el final del bloque que contiene a la expresión. Veámoslo en el siguiente fragmento de código:

```
for (   long permutacion = 1, iterador = numeroDeElementos;
```

```
            iterador > 0;
            permutacion *= iterador-- );
cout        << "Permutación de " << numeroDeElementos
            << " elementos: " << permutacion;      // OK en C++

// la siguiente declaración ocasionaría un error en C++
//  debido a la reutilización de un identificador: iterador,
//  ya definido como int
char* iterador = "Operador de la acción de iterar";
// Se causa, así, un error en C++ por múltiple declaración
```

O sea, que en realidad es como si las variables *permutacion* e *iterador* se declararan, como es usual en C, justo antes del bloque '*for*'. Y efectivamente así es: lo que C++ nos permite es clarificar la notación para permitir una más fácil lectura del código, pero sin olvidar la raíz práctica del asunto.

VALORES POR DEFECTO EN PARÁMETROS FORMALES

Todos o algunos de los parámetros formales de una función se pueden declarar por defecto. Esto es, en el momento de declaración de la función, en su definición o en ambas (si coinciden) se especificarán los valores por defecto que asumirán los parámetros deseados, de forma que cuando se produzca una llamada a la función sin especificar tales parámetros serán asumidos los notados en la declaración. Por ejemplo, la función

```
void dibujaVentana(         int x = 1, int y = 1,
                                int anchura = 20, int altura = 10 );
```

pretende dibujar una ventana en la pantalla del computador. Si no se dan argumentos en la llamada a la función, dibujará una ventana con los argumentos por defecto. Examinemos algunos ejemplos:

```
// la siguiente línea equivale a dibujaVentana( 1, 1, 20, 10)
dibujaVentana();
// la siguiente línea equivale a dibujaVentana( 2, 3, 20, 10 )
dibujaVentana( 2, 3 );
// la siguiente línea equivale a dibujaVentana( 5, 2, 12, 12 )
dibujaVentana( 5, 2, 12, 12 );
```

Si se declaran valores por defecto sólo en algunos de los parámetros formales, éstos deben significarse consecutivamente sin interrupciones hasta el final de la lista de argumentos. Por ejemplo, otra función similar a la anterior podría ser declarada como

```
void dibujaRectangulo(      int x, int y,
                                anchura = 20, altura = 10 );
```

y las llamadas a ésta podrían ser

```
dibujaRectangulo( 4, 2 );              // ( 4, 2, 20, 10 )
dibujaRectangulo( 6, 3, 17 );          // ( 6, 3, 17, 10 )
```

Hay que notar que si la declaración y la definición de la función no son coincidentes, la asignación de los parámetros por defecto únicamente habrá de ser significada una vez: bien en la declaración (o declaraciones) bien en la definición, originándose un error en caso contrario por redeclaración de tales parámetros. Esto es, pueden darse los siguientes casos:

```
// variables globales
const float proporcion = 1;
int coordenadaX = 20;
// parámetros por defecto en declaración función
double transformacionEscala( double, double = 1.0 );
// sigue declaración "normal"
void solicitaConfirmacion( int, int );
// …
double transformacionEscala( double parametro, double escala )
{
        return parametro * escala;     // escala vale 1.0 por
                                                    // defecto
                                                    // (en declaración función)
}
// parámetros por defecto en definición
//  y declaración coincidentes
float escalaProporcion( float medidaReal, float medidaPlano,
                                    float coeficiente = proporcion )
{
        return coeficiente * medidaReal / medidaPlano;
}
// parámetros por defecto en definición función
void solicitaConfirmacion( int x = coordenadaX, int y = 40 );
{
        CajaSiNo::dialogo( this, x, y );          // 'this' es un
                                                            //  puntero implícito
                                                            //  al objeto
}
```

En este ejemplo hemos utilizado variables globales (constantes o no) como argumentos por defecto de dos funciones. El uso, empero, de variables locales hubiera resultado en un error al compilar, como en el siguiente ejemplo:

```
void imprimeNodos()
{
        int miOrden = 1;
        // la siguiente definición de función incurre en ERROR
        //  en C++ 3.0, compilando, sin embargo, correctamente
        //  bajo C++ 2.0 y anteriores.
        void recorreArbolBinario(      Arbol* miArbol,
                                                    orden = miOrden );
}
```

aunque sin duda quedará más claro (o, al menos, eso espero) en el siguiente ejemplo completo:

```
#include <iostream.h>          /* para las operaciones de
                                        entrada y salida en C++ */

int main( int,char** )
{
        int local=5;           // variable local (no-global)
        void foo( int, int, int=3, int=4 );
        void foo( int, int=2, int, int);  //redeclaración OK
        void foo( int=local, int, int, int);// error en C++ 3.0

        foo( 1 );                                  // imprimiría '1234'
        foo(9,9);                                  // imprimiría '9934'
        foo(9,9,9,9);                              // imprimiría '9999'

        return 0;
}

void foo( int w, int x, int y, int z )// definición de función
{
        cout << w << x << y << z << "\n";
}
```

Tal restricción parece lógica si consideramos el propósito primario de la inclusión de parámetros por defecto como característica del lenguaje: permitir una mayor claridad del código, aislando los datos de frecuente uso (en una operación que recuerda que recuerda a la contracción de expresiones matemáticas mediante la aplicación del factor común) y permitiendo su agrupación en módulos únicos fáciles de revisar y mantener. El uso de variables locales truncaría sobremanera tal disposición. De cualquier forma -y sobre todo para aquellos lectores con dificultades para aislar los conceptos de "declaración" y "definición"- debo notar que en el ejemplo anterior he vuelto a caer en la tentación (como "reincidente autotentativo" podría ser calificado por Tweedledum y Tweedledee) de exponer un aspecto aún no visto sobre la redeclaración "correcta" de una función asignándole nuevos parámetros por defecto. Bueno, la verdad es que marcar el ritmo del relato es, lamentablemente, una prerrogativa mía frente a la paciencia del hirsuto lector. Pero, vaya, sin más dilación examinemos esta característica.

Observamos, pues, que una función dada podrá ser redeclarada para añadirle parámetros por defecto, con las únicas restricciones de no incurrir en la redeclaración de asignaciones de valores por defecto y no vulnerar la continuidad de tales asignaciones desde su aplicación en un parámetro hasta el parámetro final de la función. Veamos algunos ejemplos, comentando bajo cada línea su correctitud y, en su caso, un ejemplo de aplicación:

```
void dibujaPunto              ( int x, int y );
void distanciaAOrigen         ( int x, int y, int z );
void dibujaPunto              ( int x, int y = 40 );
// OK: ( 1 )          ==>     ( 1, 40 )
void dibujaPunto              (int x, int y = 20 );
// ERROR
void dibujaPunto              ( int x = 20, int y = 40 );
// ERROR
void dibujaPunto              ( int x = 20, int y );
// OK: ()             ==>     ( 20, 40 )
void distanciaAOrigen         ( int x = 5, int y, int z );
// ERROR
void distanciaAOrigen         ( int x, int y, int z = 10 );
// OK: ( 6, 12 )      ==>     ( 6, 12, 10 )
void distanciaAOrigen( int x, int y = 20, int z = 10 );
// ERROR
void distanciaAOrigen         ( int x, int y = 20, int z );
// OK: ( 7 )          ==>     ( 7, 20, 10 )
void distanciaAOrigen         ( int x = 5, int y, int z );
// OK: ()             ==>     ( 5, 20, 10 )
```

Nótese que no se trata aquí de la sobrecarga de la función dada, pues no se producen cambios en la signatura, identificador ni valor de retorno: únicamente se le añade una facilidad de uso. Note el lector, también, que puede encontrarse con una sorpresa si intenta chequear el anterior código en su compilador. Realmente si compila el ejemplo en Borland C++ 3.1 todas las líneas a partir de la cuarta serán flageladas como errores. ¡Vaya! ¿y por qué? -preguntará el sufrido lector. Bien, lo que ocurre es que, por ejemplo, al compilar la quinta línea, Borland C++ 3.1 arroja un error por redeclaración del valor del argumento por defecto para el parámetro 'y', pero sin embargo añade el valor por defecto 'x=20' al parámetro 'x' de la función, con lo que la siguiente línea originará un error por la redeclaración del valor del argumento por defecto para el parámetro 'x'. O sea, que el compilador asumirá parcialmente lo expresado en la línea a considerar, rechazando únicamente lo que califique como erróneo. ¿Solución? El lector puede ir cancelando con '//', de arriba hacia abajo, las líneas que originen error hasta que llegue a una correcta, para inmediatamente después volver a compilar, y así sucesivamente. Como es evidente, esto no es una característica del lenguaje.

Pero, bueno, en la práctica diaria de programación, ¿para qué sirve lo expuesto? Bien: la aplicación de tales redeclaraciones aparece evidente en la personalización de algunos módulos. Pensemos, por ejemplo, que deseamos utililzar una función de una librería comercial que dibuje una ventana típica "Acerca de ..." declarada así:

```
extern void ventanaAcercaDe( Ventana*, String, int x, int y );
```

y de la que no poseyéramos el código fuente, con lo que no podríamos de forma expedita adecuarla a nuestros gustos. Sin embargo, y en base a lo expuesto, si deseáramos aplicar unas coordenadas fijas para tales ventanas

en nuestra aplicación, podríamos redeclarar la función, en aras de la simplificación del código, de la siguiente forma:

```
extern void ventanaAcercaDe(          Ventana*, String,
                                                int x = 40, int y =20 );
// a partir de aquí el uso es transparente para el desarrollador
ventanaAcercaDe( this*, "Mi Ventana" )
```

de manera que no tendríamos que teclear repetidamente tales valores en nuestro código. La escritura y lectura del programa quedaría, así, grandemente simplificada.

La declaración de valores por defecto en parámetros formales puede llevar, sin embargo, a situaciones de ambigüedad con la sobrecarga de funciones, como podremos ver más adelante.

REFERENCIAS Y PASO DE PARÁMETROS POR REFERENCIA

En C++ se denomina referencia a un tipo derivado obtenido añadiendo el sufijo **&** a un tipo dado. Así, por ejemplo, float& se lee: "*referencia a float*".

Una variable por referencia siempre debe ser inicializada en el acto de su declaración, convirtiéndose en tal momento en un alias de su inicializador. No es legal, verbigracia, el código

```
double& referenciaANada;     // error: variable sin inicializar
```

Las referencias se comportan como variables normales del tipo al que referencian, con la particularidad de estar ligadas a la dirección de memoria de su inicializador. Esto es, de alguna forma se comportan como punteros, pero con la peculiaridad que no tienen que desreferenciarse para ser accedidas. Veámoslo en el siguiente ejemplo:

```
int original = 13;
int& referencia = original;
referencia++;
cout << original << "\n";       // salida: 14
original--;
cout << referencia << "\n";   // salida: 13
```

Como vemos, si autoincrementamos la referencia, se incrementa el original, y si autodecrementamos éste, se minora la referencia: ¡naturalmente!, pues ambos identificadores están ligados a la misma dirección de memoria.

En C++ no están permitidas las referencias a referencias, los arrays de referencias, los punteros a referencias o las referencias a *void*.

Si una referencia se inicializa con una constante, como quiera que no puede afectarse a una dirección de memoria, el compilador crea un tipo temporal. O sea:

```
int& numeroCabalistico = 7;
```

es tratado por el compilador de la forma

```
int enteroTemporal = 7;
int& numeroCabalistico = enteroTemporal;
```

Así mismo, la asignación de un inicializador con tipo distinto al apuntado por la referencia genera, también, una variable temporal. Vemos, pues, que

```
float unidad = 1.0;
int& floatTruncado = raizDeDos;
```

equivale en la práctica a

```
int enteroTemporal = int( unidad );       // cast funcional
int& floatTruncado = enteroTemporal;
```

Las referencias son usadas como parámetros formales de funciones por las mismas razones que se usan los punteros: para permitir la modificación de los datos en el ámbito local de las funciones y para evitar la penalización en tiempo de ejecución que supone la copia de los argumentos en las llamadas a funciones con argumentos pasados por valor. Examinemos el siguiente ejemplo:

```
void intercambiaDatos( int& primero, int& segundo )
{
        int temporal = segundo;
        segundo = primero;
        primero = temporal;
}
// ...
int primerParametro = 1;
int segundoParametro = 2;
intercambiaDatos( primerParametro, segundoParametro );
cout << primerParametro;     // salida: 2
cout << segundoParametro;  // salida: 1
```

Vemos que, en comparación con la sintaxis de punteros, el código local a la función del ejemplo es más natural e intuitivo, al no necesitar operadores de desreferenciación. Advertimos, también, que aunque la función se declaró con dos parámetros "referencias a entero", en la llamada a ésta se aplican variables de tipo 'int' (y no int&, produciéndose una conversión trivial), de forma que se produce un "paso" de tales variables "por referencia" de una forma similar a la conocida por los lectores con experiencia en Pascal.

Aparte de todo lo anterior, las referencias son insustituibles en la inicialización por asignación de constructores de clases, como ya veremos, en ese continuo "huir hacia adelante" en que se está convirtiendo esta serie.

EL ESPECIFICADOR "*INLINE*"

Este especificador debe ser utilizado precediendo a la declaración de una función, y constituye una sugerencia al compilador para que éste sustituya "*en línea*" cualquier uso de la función por el cuerpo de ésta, en lugar de efectuar la llamada de función. Se logra, así, evitar la penalización en tiempo de ejecución determinada por la llamada a la función.

Dado que se trata de una recomendación, el compilador podría ignorar la especificación **inline** en una función por muy distintas razones: por ser recursiva, por ser demasiado larga, por contener una instrucción *goto*, etc. En general estos detalles dependen de la implementación del compilador. En Visual C++ se afirma que el mecanismo de "inlining" no posee restricciones: o sea, toda llamada a una función declarada como "inline" será sustituida por su cuerpo, sin excepción alguna. En realidad esto no tiene por qué ser una ventaja: en ARM se establece la accesoreidad del uso de tal especificador.

Hay que notar, ant todo, que una función **inline** no es igual a una macro del preprocesador, pues en éstas no se produce chequeo alguno de tipos. Se obtiene, pues, "*lo mejor de dos mundos*". Conviene destacar, por otro lado, que el especificador no surtiría efecto sobre una declaración de función, pues no habría código que sustituir.

El especificador **inline** está especialmente indicado para optimizar funciones pequeñas, de pocas líneas y frecuente uso. La eficacia de este cualificador puede ser comprobada -es un decir- en el siguiente código:

```
void funcionNormal() {};
inline void funcionInline() {};
void chequeaFuncionInline()
{
        Cronometro cronometro;
        cronometro.empieza();
        for (long contador = 1; contador < 10000000; contador++)
                funcionNormal();
        cronometro.imprime();
        cronometro.empieza();
        for (long contador = 1; contador < 10000000; contador++)
                funcionInline();
        cronometro.imprime();
        cronometro.cierra();
}
```

En la anterior función se hace uso de un objeto cronometro, que se declara localmente y se inicializa antes de cada bucle imprimiendo el lapso de tiempo transcurrido seguidamente. Aunque el resultado

depende de la implementación, el bucle conteniendo la función **inline** será ejecutado una media de *4 veces* más rápido. El lector podría chequear su sistema sustituyendo lo relacionado con el objeto cronometro por un más simple contador horario.

¿Por qué no declarar entonces, visto lo anterior, todas las funciones como **inline**? Pues porque, en palabras del Dr. Stroustrup, el mecanismo de "**inlining no es una panacea**". El abuso de esta especificación obligaría, por ejemplo, a recompilar todos los módulos en que aparecieran funciones **inline** cuando éstas sufrieran alguna modificación. El hecho, por otra parte, de que el compilador deba mantener en memoria el código de las funciones **inline** puede ocasionar colapsos por falta de memoria en algunas implementaciones C++. El tamaño del código puede aumentar extraordinariamente y se debe recordar, por último, que la definición de funciones **inline** en los archivos de cabecera acarreará la publicitación del código fuente de las mismas.

SOBRECARGA DE FUNCIONES

C++ permite el uso en el mismo ámbito de igual nombre de identificador referido a funciones con distintos argumentos. La llamada a la función apropiada es resuelta por el compilador, de forma que se produce una deseable homogeneización del código, como ya vimos al describir el *polimorfismo* en un capítulo anterior.

La sobrecarga de funciones se origina, simplemente, al declarar una función, anteriormente ya declarada, con distinto número y/o tipo de argumentos, no afectando al tipo de retorno de las funciones. De esta manera tenemos que dada, por ejemplo, la función

```
long multiplicar( int& multiplicando, int multiplicador );
```

las siguientes declaraciones serían calificadas como errores por *redeclaración* en tiempo de compilación, pues difieren de la primera únicamente en el tipo de retorno o en tipos asimilables por conversiones triviales:

```
void multiplicar( int, int );                          // error
long multiplicar( int variable1, variable2 );          // error
long multiplicar( int, int& );                         // error
// recordemos que un 'typedef' no es un tipo separado
// sino simplemente un 'sinónimo' de un tipo[6]
```

[6] ¡Atención a esta característica! A pesar que la afirmación es rigurosamente exacta, el siguiente código (y otros parecidos) podrá ser compilado sin problemas con Borland C++ 3.X y 4.0, mientras que AT&T C++ 3.0 -de acuerdo con lo establecido en ARM- flagelaría como errores las dos últimas líneas:

```
int pruebaBorland( int );
typedef int Entero;
```

```
typedef int Entero;
long multiplicar( Entero, Entero );                    // error
```

mientras que las siguientes declaraciones serían reputadas como correctas (nótese que algunas de tales funciones son ellas mismas también sobrecargadas correctamente), siempre que tengamos la precaución, como vimos cuando repásabamos los parámetros por defecto, de no compilarlas junto con las anteriores, a fin de evitar la posibilidad de "asunciones parciales" por parte del compilador:

```
float multiplicar( float, float );
long multiplicar( int, int, int=1 );
long multiplicar( long, int );
double multiplicar( double, double, double);
short multiplicar( short, short );
void multiplicar();
long multiplicar( volatile int&, int );
long multiplicar( int&, const int& );
long& multiplicar( int, int* );
long& multiplicar( int, const int* );
char* multiplicar( char, int );
char* multiplicar( int, char );
// Atención: 'Entero' es un tipo distinto de 'int' en C++
enum Entero ( cero, uno, dos, tres ); C++
long multiplicar( Entero, int );
```

Notemos que, si bien son distintos, los prototipos siguientes:

```
long multiplicar( int, int );                  // función original
long multiplicar( int, int, int=1 );// función sobrecargada
```

causarán que la primera de las siguientes llamadas concretas a la función

```
multiplicar( 2, 1 );              // error: ambigüedad
multiplicar( 2, 1, 1 );                    // ok
```

sea declarada como error, pues encaja con cualquiera de los dos prototipos. Aquí surge la cuestión sobre la idoneidad de tal sobrecarga, resultando así que o bien se añade el tercer argumento al prototipo de la función original,

```
int pruebaBorland( Entero );
Entero pruebaBorland ( int );
```

Se trata aquí de una cuestión de adaptación del compilador a los estándares del lenguaje. Debemos recordar que, aun siendo Borland C++ 4.0 la implementación de C++ para PC's más ajustada a los "estándares", ésta se basa sólo parcialmente en AT&T C++ 3.0 y en el borrador del estándar de C++ proveniente de X3J16. Deberemos esperar a futuras versiones para ver si se solucionan estas "curiosidades". Mi consejo: el lector deberá evitar las construcciones del tipo expuesto, pues de otra manera su código en el futuro podría no ser portable.

eliminando la sobrecarga, o bien se puede suprimir el parámetro formal por defecto, eliminando la ambigüedad en la concrección.

En el caso de que en una llamada a una función sobrecargada no se produzca una correspondencia **exacta** con el tipo de cada uno de sus argumentos, se buscará la mejor de las correspondencias posibles con los prototipos disponibles aplicando las siguientes reglas ordenada y consecutivamente a cada uno de los argumentos de la función llamada:

1º) Conversión trivial: tipo pasa a tipo&, tipo& pasa a const tipo&, tipo& pasa a volatile tipo&, tipo* pasa a const tipo* y tipo* pasa a volatile tipo*.

2º) Promoción: los tipos char, unsigned char, short int e int son promocionados a int si int puede contenerlos, o a unsigned int de otra manera; float pasa a double y double pasa a long double.

3º) Conversión standard: cualquier tipo numérico pasa a otro tipo numérico; las enumeraciones pasan a tipos numéricos; el "cero" se convierte en instanciación de puntero a cualquier tipo o de tipo numérico; un puntero a un tipo determinado se convierte en un puntero a void*; los punteros, referencias y objetos de clases derivadas públicamente se convierten en punteros, referencias y objetos a clases base de la jerarquía; un puntero de una clase base se convierte en puntero a una de sus clases derivadas públicamente.

4º) Conversión definida-por-el-usuario: si se trata de un tipo definido por una **class** y en ésta se ha implementado una función miembro o *friend* de la forma operator tipo(), denominada *operador de conversión*, se aplicará el método por ella definido al tipo del argumento y se buscará la correspondencia del nuevo tipo en los prototipos de la función sobrecargada.

5º) Correspondencia con elipsis: un argumento de cualquier tipo establecerá correspondencia con una función prototipada con lista de argumentos (...).

Hay que notar que si bien los intentos de ajustar un valor a un tipo de argumento formal se dan mediante la aplicación ordenada de los diferentes formatos de conversión, no existe prevalencia de conversión dentro de cada uno de éstos, de forma que debemos desechar la idea intuitiva de que el compilador efectuará la conversión que requiera menos esfuerzo o tiempo: en realidad el compilador probará todas las posibilidades dentro de cada uno de los cinco tipos de conversión, declarando un error de ambigüedad cuando sea posible realizar más de una correspondencia entre valor y tipo de argumento. No existe, tampoco, precedencia en la conversión en razón del orden de declaración de las funciones sobrecargadas. Así, por ejemplo, dadas las siguientes declaraciones:

```
extern int funcion( int );
extern int funcion( short );
```

serán declaradas como errores las llamadas:

```
funcion( 1L );          // error: long pasa a int
                                // ó a short indistintamente
enum boolean { FALSO, VERDADERO };
funcion( FALSO );    // error: enum pasa a int
                                // ó a short sin precedencias[7]
```

Esto es, aplicando las conversiones estándar detalladas anteriormente, los valores de tipo *long* y *enum* se convertirán en cualquier otro tipo numérico para ajustarse a los tipos de los parámetros formales de las funciones sobrecargadas -en este caso *int* o *short*-, de forma que, al darse más de una posibilidad, se señala como error por ambigüedad la llamada concreta.

En las versiones anteriores a la AT&T C++ 3.0 se observaba una regla de precedencia en la aplicación de conversiones de ajuste: las conversiones que requerían la aplicación de variables temporales erán consideradas de un orden inferior al de las conversiones que no las requerían, tomando éstas precedencia sobre aquéllas. Así, dadas

```
void hazNoSeQue( long );
void hazNoSeQue( char& );
```

en el siguiente código:

```
int numeroDeOjos = 1;                    // Ojos de un cíclope
char letraErotica = 'S';          // Desfasada calificación moral
// la siguiente llamada será calificada como AMBIGÜA
//  bajo C++ 3.0: no hay precedencia de conversión.
hazNoSeQue( numeroDeOjos );          // OK bajo C++ 2.0:
                                                // llama a hazNoSeQue( long )
hazNoSeQue( letraErotica );   // OK: llama a hazNoSeQue(char&)
```

observamos que dado que la conversión de **int** a **char&** requiere el uso de una variable temporal, la conversión a **long** hubiera tomado precedencia bajo C++ 2.0 y se hubiera producido una llamada sin ambigüedad a la función con tal argumento. En C++ 3.0, sin embargo, se produciría error por ambigüedad en la resolución de la sobrecarga.

Se aplicarán reglas de precedencia, igualmente, en la conversión de objetos, referencias o punteros de clases derivadas con carácter público a objetos, punteros o referencias a sus clases base, en razón de la proximidad jerárquica de las clases involucradas en la conversión.

[7] Surge aquí de nuevo lo ya apuntado en la nota anterior: esta línea compilará sin error con Borland C++ 3.X, aunque sí fallará en el cfront 3.0 de AT&T. Afortunadamente Borland C++ 4.0 ha subsanado este desajuste, y correctamente origina un error en compilación por ambigüedad en tal línea.

Cabría preguntarse: ¿Por qué esta parafernalia de conversiones? ¿Por qué no aplicar reglas más simples, como la del mínimo esfuerzo? Realmente el grueso de las reglas conviene a las conversiones standard en C y también, por compatibilidad, en C++, pues los ajustes definidos por los operadores de conversión en las clases, propios de C++, se limitan a uno por tipo. La mejor manera de evitar ambigüedades es, en lo posible, evitarlas: si deseamos evitar problemas con un determinado tipo, debemos implementar una función sobrecargada que acepte ese tipo exacto de argumento.

El lector podrá encontrar una descripción exhaustiva de las cinco reglas de conversión en sobrecarga de funciones en el apartado 13.2 de ARM.

LOS OPERADORES "*NEW*" Y "*DELETE*"

Las operaciones de manejo de la memoría libre se realizan en C++ mediante los operadores **new** y **delete**.

El operador **new** se utiliza para alojar en memoria un objeto de tipo predefinido o definido-por-el-usuario (instanciación de una clase), reservando primero la suficente cantidad de memoria de almacenamiento libre, inicializándolo depués y devolviendo, al fin, un puntero al mismo. De esta forma la creación dinámica de un array de char se realizaría de la forma:

```
const MAX_ARRAY = 100;
char* punteroAArrayDeChar, pc;
pc = punteroAArrayDeChar = new char[ MAX_ARRAY ];
*pc = 'W';
```

Veamos también, de igual manera, otras posibilidades sintácticas igualmente válidas:

```
int* pEntero1, pEntero2;
pEntero1 = new int;          // construye un objeto de tipo int
pEntero2 = new( int );       // notación funcional
                                    //  equivalente a la anterior
int OK = ( pEntero1 != pEntero2 );      // cierto SIEMPRE;
```

El operador global **::new** aparece originalmente declarado de la siguiente forma:

```
void* operator new( size_t tamanoEnBytesDelObjetoAAlojar );
```

y el hecho de que en los ejemplos no hayamos tenido que explícitamente declarar el tamaño en bytes de la memoria libre requerida se debe a que es el sistema el que automáticamente se responsabiliza de tal cálculo y asignación (de la misma forma que en el álgebra de punteros). Debido a tal especial circunstancia cualquier sobrecarga de este operador debe devolver **void*** y poseer un primer argumento de tipo **size_t** (typedef establecido en

"stddef.h") representando el tamaño en bytes a alocar por el sistema. El lenguaje establece, como característica estándar[8], la siguiente sobrecarga predefinida de **::new**:

```
void* operator new(          long tamanoEnBytesDelObjetoAAlojar,
                                    void* direccionDeMemoria );
```

la cual nos permite aplicar un área de memoria predeterminada al alojamiento del nuevo objeto:

```
// asignación estándar de ::new
char* peliculaDeHoy = new char[ 30 ];
// seguidamente deseamos aprovechar el almacenamiento libre
//  asignado al título de la película de hoy
//  para el almacenamiento del título de la película de mañana.
char* peliculaDeManana = new( peliculaDeHoy ) char[ 30 ];
// seguidamente se "nulifica" el puntero "peliculaDeHoy"
//  para evitar futuras e indudablemente peligrosas
//  desreferenciaciones
peliculaDeHoy = 0;
```

Nótese, de nuevo, que en el ejemplo sólo hemos proporcionado como argumento el puntero a la dirección de la memoria libre a reutilizar, ocupándose de la determinación del tamaño del espacio apuntado por el identificador peliculaDeHoy el propio sistema. El desarrollador sería en este caso responsable de la correspondencia entre el tamaño del objeto a almacenar y la memoria disponible para su "reutilización".

Cuando el operador **new** se utiliza para el alojamiento de instancias de clases (objetos), como por ejemplo

```
ClaseEjemplo *punteroAObjetoDeClaseEjemplo = new ClaseEjemplo;
```

se pone en marcha el siguiente esquema secuencial:

1) Se busca una definición sobrecargada del operador **new** en la clase del objeto a crear (en este caso ClaseEjemplo), de la forma

```
void* ClaseEjemplo::operator new( size_t ) {
        // aquí vendría el código apropiado
}
```

y se ejecuta el código de la misma.

[8] El adjetivo estándar relativo al lenguaje C++, como se ha advertido varias veces, se refiere a las características derivadas de las implementaciones de AT&T. En el presente caso la sobrecarga del operador ::new aparece predefinida en el archivo "new.h", mientras que en otras implementaciones pudiera no existir tal declaración, debido, entre otras cosas, a la facilidad con que tal sobrecarga puede ser implementada por el propio desarrollador.

2) Si la definición anterior no se encuentra en la clase del objeto, se procede a su búsqueda en las clases de las que ésta públicamente deriva, para seguidamente ejecutarla. Como más adelante veremos en la derivación de clases, esta característica originará la aplicación de un esfuerzo adicional en el diseño de la sobrecarga del operador en esquemas de derivación pública.

3) Si no se encuentra ninguna redefinición del operador new, entonces se aplicaría el operador global **::new()**, que invocaría el constructor apropiado para el nuevo objeto (en este caso, el constructor por defecto: ClaseEjemplo::ClaseEjemplo()), asignando seguidamente un puntero al mismo al puntero punteroAObjetoDeClaseEjemplo.

Cuando lo que se desea crear es, por el contrario, un array de objetos, como en el código siguiente:

```
ClaseEjemplo *punteroAArrayDeObjetos = new ClaseEjemplo[numero];
```

entonces se produciría únicamente una llamada al operador global **::new()**, que invocaría el constructor apropiado para cada uno de los nuevos objetos, desde ClaseEjemplo[0] hasta ClaseEjemplo[numero-1], para después asignar un puntero a tal array al puntero punteroAArrayDeObjetos.

Si tenemos en cuenta la comparación, por otra parte siempre presente, entre tipos incorporados y tipos definidos-por-el-usuario (cuales son las clases), encontramos que en el caso de arrays de tipos predefinidos (int, char, etc.) es el entorno del lenguaje el que "recuerda" el tamaño del array declarado. De igual lógica forma, a partir de AT&T 2.1, es el entorno C++ el encargado de guardar el tamaño de un array de objetos. En versiones anteriores, empero, el desarrollador debía cargar con tal responsabilidad.

Hasta ahora hemos visto la creación de nuevos objetos por medio de los constructores por defecto de las clases de las que serían instanciaciones. La creación de nuevos objetos usando otros constructores puede realizarse mediante la sintaxis:

```
ClaseEjemplo* punteroAObjetoClaseEjemplo =
                                    new ClaseEjemplo( argumento1, ... );
```

En este caso se invocaría el constructor adecuado a la lista de argumentos provista.

Debe recordarse que el operador **new** intenta alojar el nuevo objeto en el área de memoria de almacenamiento libre: si no existiera suficiente memoria para el alojamiento del objeto, el operador global **::new** devolverá **cero** (0). El desarrollador es responsable, pues, de generar código de chequeo del resultado de la aplicación del operador. El sistema, en realidad, utiliza el siguiente mecanismo : si **::new** falla en el alojamiento de un objeto (por haberse agotado el almacenamiento libre), se produce el chequeo de un

manipulador predefinido del tipo "puntero a función sin argumentos y con valor de retorno void" denominado **_new_handler**, declarado en el archivo de cabecera standard **"new.h"** de la siguiente forma:

```
extern void( *_new_handler ) ();
_new_handler = 0;              // asignación a CERO por defecto
```

de forma que si tal manipulador apunta a cero, **::new** devuelve cero, mientras que, en caso contrario, se produce una llamada a la función apuntada por **_new_handler**. Podemos, pues, suprimir el chequeo directo del resultado del operador **::new** mediante la reasignación al manipulador de la dirección de una función destinada al manejo de los errores de alocación de objetos en memoria. Tal asignación la podemos realizar bien directamente

```
extern void errorPorMemoriaLibreAgotada();
_new_handler = errorPorMemoriaLibreAgotada;
```

bien a través de una función específica para ello declarada, también, en **"new.h":**

```
void (* set_new_handler( void ( * )() ) )();
```

de la siguiente forma (siguiendo el ejemplo anterior):

```
set_new_handler( errorPorMemoriaLibreAgotada );
```

Pero, ¿qué ocurre tras la ejecución de la función apuntada por **_new_handler**? Pues bien, el sistema asume que se ha solucionado el error de alocación generador de la llamada y si explícitamente no se codifica en la función apuntada por el manipulador una llamada a, por ejemplo, la función estándar **exit()** (incluida en el archivo C "stdlib.h"), al devolver el control al operador global **::new** éste intentará realizar de nuevo el alojamiento en memoria, fallará en el intento, llamará de nuevo a la función apuntada por el manipulador y vuelta a empezar, originando posiblemente un bucle infinito equivalente a una condición de error irrecuperable. Por supuesto una sobrecarga adicional del operador **new** podría manejar de forma más adecuada esta situación. Como quiera, por otra parte, que estos errores se producen únicamente en tiempo de ejecución, quizá uno de los aspectos más interesantes de esta técnica lo constituya la posibilidad de implantar subrepticiamente un sistema parcial de "captación de excepciones" que podría ser utilizado, como veremos, en el chequeo de constructores de clases.

Pareja dicotómica del operador new, el operador **delete** se usa, en contra-partida, para la restitución al área de almacenamiento libre de la memoria anteriormente utilizada por mediación de **new**. Su sintaxis es transparente, aplicada sobre punteros a objetos almacenados mediante **new**:

```
// Se declaran distintos punteros
int* punteroAInt;
float* punteroAArrayDeFloat;
ClaseEjemplo* punteroAObjectoDeClaseEjemplo;
ClaseEjemplo* punteroAArrayDeObjetosClaseEjemplo;
// …
// Seguidamente se aplica el operador new
punteroAInt = new int;
punteroAArrayDeFloat = new float[ 3 ];
punteroAObjetoDeClaseEjemplo = new ClaseEjemplo;
punteroAArrayDeObjetosClaseEjemplo = new ClaseEjemplo[ 6 ];
// ..
// Se procede al desalojo de la memoria utilizada
//  por las variables y arrays
delete( punteroAInt );
delete[] punteroAArrayDeFloat;// delete[3] punteroAArrayDeFloat
                                                    //en AT&T 2.0
delete punteroAObjetoDeClaseEjemplo;
delete[] punteroAArrayDeObjetosClaseEjemplo;// delete[ 6 ] ...
                                                    // en AT&T 2.0 y anteriores
```

El operador global **::delete** aparece originalmente declarado como

```
void operator delete( void* punteroAMemoriaADesalojar );
```

incorporándose al estándar del lenguaje también la siguiente sobrecarga:

```
void operator delete(       void* punteroAMemoriaADesalojar,
                                size_t tamanoEnBytesAreaMemoria );
```

responsabilizándose el sistema (como en el caso del operador **new**) de la inicialización del segundo argumento, que representa el tamaño de la memoria a desalojar. De igual manera, las sobrecargas de este operador deberán devolver void y significar el menos un primer argumento de tipo void*, observándose también que si se aplican más argumentos el segundo de ellos habrá de ser forzosamente de tipo size_t y para uso del sistema.

De forma pareja a como ocurre con el operador **new**, cuando se aplica el operador **delete** a un objeto de tipo definido-por-el-usuario, como en el ejemplo

```
delete punteroAObjetoDeClaseEjemplo;
```

en primer lugar se busca una posible implementación de, por ejemplo:

```
void ClaseEjemplo::operator delete( void* ) {
        //aquí vendría el código
}
```

en la clase del objeto; seguidamente, si no se ha encontrado, en las clases base públicas de la clase actual del objeto; por último, si no se ha sobrecar-

gado el operador **delete**, se aplica el operador global **::delete()**, el cual invocará el destructor apropiado para el objeto (en este caso, el destructor por defecto ClaseEjemplo::~ClaseEjemplo()).

Vemos que la sintaxis varía en el caso de tener que liberar la memoria ocupada por arrays. En tal supuesto el código

```
delete[] punteroAArrayDeObjetosClaseEjemplo;
```

simplemente aplica el operador global **::delete()** al array, invocando el destructor apropiado para cada uno de los objetos, desde punteroAArrayDeObjetos-ClaseEjemplo[0] hasta punteroAArrayDeObjetosClaseEjemplo[5]. Como vimos anteriormente, es responsabilidad del sistema el mantenimiento del tamaño del array, de forma que no tenemos que explicitarlo en la sintaxis del operador.

En versiones anteriores a C++ AT&T 2.1 el compilador exigía el tamaño del array al que se habría de aplicar el operador **delete**, debiendo codificar lo siguiente:

```
delete [ 6 ] punteroAArrayDeObjetosClaseEjemplo;
```

Esta codificación es soportada por C++ como un anacronismo y actualmente origina, a lo sumo, un aviso o *warning* del compilador, aunque podría perjudicar la portabilidad del código a futuras versiones de C++.

Es importante notar, en aras de la claridad conceptual, que sobre un array de objetos de tipo incorporado o no, siempre actuarán los operadores globales **::new** y **::delete**, pero que sobre un objeto de tipo ArrayDeClasesEjemplo definido, por ejemplo, así:

```
Class ArrayDeClasesEjemplo {
private:
        ClaseEjemplo** punteroAArrayDeClasesEjemplo;
        //...
}
```

la aplicación de los operadores será la siguiente:

```
ClassArrayDeClasesEjemplo* punteroAObjetoArray;
punteroAObjetoArray = new ClassArrayDeClasesEjemplo;
delete punteroAObjetoArray;
```

pues no se trata aquí de un array de objetos, sino de un objeto con características de array. Se buscarán primero, por tanto, posibles sobrecargas de los operadores antes de aplicar los operadores globales. Se puede comprender ahora que el compilador trata a un array como un objeto incorporado de agregación de objetos y que, como tal, no redefine los operadores **new** o **delete**, debiendo aplicar en las operaciones con éste los operadores globales. Persiste, pues, la homogeneización entre tipos predefinidos y clases.

Es conveniente añadir que el sistema, a pesar de mantener o "recordar" el tamaño de los arrays de objetos, no es "inteligente" con respecto a los identificadores de los arrays. Esto es, la aplicación del operador **delete** requerirá siempre[9] (en el caso de arrays) de la sintaxis **[]**, que indicará al entorno que se pretende "destruir" un array más que un objeto. Supuestas las declaraciones anteriores, si ejecutamos la siguiente línea

```
delete punteroAArrayDeObjetosClaseEjemplo;
```

lo que se generará es una llamada al constructor ClaseEjemplo::~ClaseEjemplo() para el objeto *punteroAArrayDeObjetosClaseEjemplo (o, lo que es lo mismo, punteroAArrayDeObjetosClaseEjemplo[0]), pero quedarán sin destruir los objetos restantes del array (desde punteroAArrayDeObjetosClaseEjemplo[1] hasta punteroAArrayDeObjetosClaseEjemplo[5]).

El operador global **::delete** ha de aplicarse a punteros a objetos cuyo alojamiento haya sido procurado por el operador global **::new**, pues de otra forma el resultado será indefinido. El lenguaje asegura, por otra parte, la aplicación del operador **::delete** a punteros apuntando a cero (NULL) como una operación siempre válida.

Por supuesto los operadores globales son siempre accesibles por medio del operador **::**, de forma que pueden de esta manera ser invocados en sustitución del posible operador sobrecargado que en razón de las reglas de ámbito conviniera aplicar.

MIGRACIÓN DE ANSI C A C++: REGLAS BÁSICAS

Ya hemos repasado las diferencias básicas entre ANSI C y C++, de forma que tenemos una suerte de breviario para procurar la transición de uno al otro lenguaje. Dadas las características de eficacia, funcionalidad y soporte de OOP que provee C++ es lógico pensar que en los próximos años se producirá una migración masiva desde sistemas C hacia C++. No sobran, sin embargo, algunas reglas elementales que, sin duda, tornarán más amable y suave tal adaptación:

- El propio Bjarne Stroustrup ha afirmado que C++ no es la medida de todas las cosas: existen problemas que no tienen una buena solución en C++. Por esto es necesario, ante todo, determinar qué partes de un sistema software son susceptibles de mejora mediante el uso de las características de este nuevo lenguaje, para inmediatamente evaluar los beneficios del nuevo desarrollo contra los costos del mismo. Si un sistema funciona bien no hay ninguna razón válida para cambiarlo.

[9] Con la única excepción de arrays unidimensionales (vectores) que carezcan del operador **delete** y de destructor, como, por ejemplo, los vectores de tipos incorporados (*int, float*, etc.)

- No debe mezclarse indiscriminadamente el código C++ con el existente en C, pues esto normalmente causará más problemas que beneficios: el mantenimiento se tornará aún más costoso y el desarrollo en C++ se verá sin duda constreñido por su forzado interfaz con C. Debe mantenerse, pues, una limpia separación entre las implementaciones de ambos lenguajes, lo que no quiere decir que no se utilicen en C librerías y características de C++; más bien lo que se sugiere es que la incorporación de C++ sea modular e incremental.

- El uso por C de librerías de C++ debiera realizarse siempre, en lo posible, a través de funciones C compiladas en los módulos C++, las que, a su vez, accederían a las funciones y métodos de la librería C++. De esta forma se evitarán problemas de portabilidad derivados de las implementaciones específicas de la codificación interna (*name mangling*) por el enlace de tipo-seguro. Así, por ejemplo, podría definirse un archivo C++ que sirviera de interface a C para el uso de funciones de algebra de matrices de la siguiente forma:

```
extern "C" int determinante( Matriz* miMatriz) {
        return miMatriz.determinante();
}
```

de esta forma podría llamarse sin problemas desde cualquier archivo C a la función determinante declarándola de la siguiente forma:

```
extern determinante( void* );
```

- Existen funciones de la librería estándar de C más eficientes, para determinados propósitos, que las equivalentes en C++. Así, por ejemplo, pudiera elegirse **printf()** en lugar de **cout.operator<<(...)** en una aplicación específica. No es C, pues, el "hermano pobre" de C++ ni tampoco éste es el verdugo de aquél. Debe elegirse siempre por tanto, en consonancia con el espíritu del nuevo lenguaje, la implementación más efectiva.

5

DONDE "LA CLASE" SE EVIDENCIA

En el presente capítulo abordaremos formalmente la clave del acercamiento de C++ a la OOP: *las clases*. Alrededor de ellas gira la práctica totalidad de las técnicas del nuevo paradigma de programación, de tal forma que focalizan el tratamiento de las nuevas características del lenguaje. Empezaremos, por decirlo así, con algo de teoría, algo de comentario y un tanto más de ejemplos, de tal forma que no nos complicaremos demasiado con el "qué" y nos centraremos más en el "cómo". O sea, que se podría escribir y escribir sobre las características del lenguaje relacionadas con las clases, pero lo cierto es que eso ya está hecho ("El Manual de Referencia de C++ Anotado", cariñosamente conocido como "ARM" es un libro -indispensable, por otro lado- dedicado, prácticamente, sólo a ello). En definitiva, intentaremos asir el concepto que se esconde tras "*la clase*" para así poder usarla con juicio. ¿Con juicio? Bueno, en la comunidad C++ circula una historia, a estas alturas ya muy deformada, que cuenta cómo un equipo de desarrollo que hasta entonces trabajaba en C, con su C++ flamante y sin estrenar, tuvo que acometer su primer proyecto en el nuevo lenguaje: los programadores se dijeron "¿para qué analizar o diseñar? ¿No estamos en C++? ¡Hagamos lo que sea, y de seguro que será OOP!". Al cabo de seis meses se encontraron en que, virtualmente trabajando cada uno por su lado, habían desarrollado la friolera de más de 3.000 clases, cada una de ellas repleta de código usando casi todas las características posibles del lenguaje, a cual más "inservible". Lo cierto es que tardaron pocas horas en decidir el arrinconamiento de este ingente trabajo y volvieron a comenzar el proyecto en puro C, jurándose ciertas barbaridades y un reparto de odios eternos a ciertos "gurús" de la OOP. Esto sucedió en USA, por supuesto, y lo cierto es que el nombre de la compañía ha quedado grabado en fuego en la breve historia de este lenguaje, al igual que sucedió con el causante del incendio en la Biblioteca de Alejandría. Lo mejor es evitar el nombre en ambos casos. ¿La moraleja? Ahí va: "*la clase*" está reñida con lo superfluo y la tontería (y lo cierto es que esto lo podría haber muy bien proclamado el mismísimo Lord Brummel). Cuando se aprende un nuevo lenguaje de programación, inmediatamente se intentan aplicar todas las técnicas estudiadas, lo que constituye una barbaridad pareja a la de, tras un curso de

Cocina, intentar cocinar siempre con la totalidad de las viandas: imaginen el *pastiche*. Así que ... ¡tranquilidad! y seamos prácticos. Pero, antes de entrar en materia, y aun a riesgo de resultar dolorosamente trivial, vamos a echarle un vistazo a una materia "opinable" por naturaleza: el estilo. ¿Y por qué? Bien, resulta contraproducente que C++ pretenda clarificar mediante distintos mecanismos el aspecto del lenguaje y que éste, sin embargo, aparezca como un plato de tagliatelli demasiado hervidos. Fiense de mi brevedad.

ALGUNAS NOTAS SOBRE ESTILOS DE CODIFICACIÓN

Existen en la actualidad, básicamente, tres diferentes estilos puros de indentación del código C (y, por ende, C++), distinguiéndose significativamente en la colocación de las llaves (**{}**) que encierran bloques de código tales como los de funciones, estructuras de control, bucles, etc.

Por un lado, y en primer lugar, está el viejo y compacto estilo *Kernighan-Ritchie*, que abre llave al final de la línea de la cabecera de la estructura de control y la cierra bajo ésta y a su nivel, indentando el código un nivel.

```
switch ( respuesta ) {
case NO:
        cout << "La respuesta es NO";
default:
        cout << "No sabe. No contesta";
}
```

Otro estilo abre llave justo debajo de la cabecera de la estructura de control indentándola un nivel, formando vertical con el código del bloque y con el cierre de la llave.

```
for ( int iterador= 0; iterador < 10; iterador++ )
        {
        cout << iterador;
        }
```

El último estilo, al fin, abre llave justo debajo de la cabecera de la estructura de control y a su nivel, indenta el bloque un nivel y cierra llave al nivel de la apertura.

```
void Empleado::estableceNombre( char* cadenaNombre )
{
        nombre = cadenaNombre;
}
```

Dado que la elección de estilo de indentación es una materia de gusto personal y de claridad en la lectura del código, mi modesto consejo es que, una vez escogidas las directrices que harán más legible nuestro código -sean las que sean- deberán ser empleadas sin lagunas. O sea, que no debe

cambiarse de estilo en cada módulo; ni siquiera en cada programa. En lo que a mí respecta y como el lector ya habrá podido apreciar, uso una mixtura propia y coherente de los estilos anteriores, conservando el estilo *Kernighan-Ritchie* para las sentencias de control (*for, switch, do-while*, etc.), mientras que utilizo el tercer estilo (llaves al nivel de la estructura y código indentado un nivel) para las funciones, métodos, constructores y destructores. Veámoslo:

```
void imprimePares( long limiteSuperior )
{
        for ( long i = 1; i <= limiteSuperior; i++ ) {
                if ( i & 2 == 0 )
                        cout << i << "\n";
        }
}
```

Prefiero, por otro lado, anteponer el tipo de retorno al nombre de la función en la misma línea

```
int main( int, char** ) {}
```

en lugar del también aceptable estilo

```
int
main( int, char** ) {}
```

Ahora un tema que desata polémicas: cuando se declaren distintas variables en una línea (algo, por otra parte, usualmente del todo innecesario y que siempre es mejor evitar[10]), normalmente adscribiré el operador de *puntero* (*****) al identificador de la variable en lugar de al del tipo, para evitar errores de lectura. Así puede verse claramente que

```
char *cadena, letra;
```

declara un puntero a char (cadena) y un simple carácter (letra), mientras que esta otra sintaxis:

```
char* cadena, letra;
```

podría inducir a pensar que se están declarando dos punteros a char (los espacios no son considerados por el analizador léxico del compilador).

[10] Piénsese que el compilador **ignora**, a efectos de optimización del código, el hecho de que unas determinadas variables hayan sido declaradas o no en la misma línea. La manía economizadora no proporciona, pues, a excepción de en algunos ejemplos triviales, ninguna ventaja apreciable, procurando, en la mayoría de los casos, una dificultad adicional para el lector. Mi consejo: ¡eviten las multi-declaraciones en una línea! ¡Reserven el ingenio para la concisión de los algoritmos!

Cuando no haya lugar a confusión se empleará indistintamente cualquiera de las dos notaciones, en razón de conseguir la máxima inteligibilidad posible en cada contexto. Lo cierto es, sin embargo, que yo siempre suelo utilizar el operador de puntero adscrito al identificador del tipo (debido, sin duda, a la gran cantidad de veces en que no he tenido más remedio que hacerlo así en los prototipos de funciones sin variables de "maquillaje"). También es cierto que para mí no hay confusión posible en la lectura: *magister dixit*.

En cuanto a la codificación de identificadores suelo usar el estilo de Smalltalk (palabras seguidas sin separación e iniciadas por letras mayúsculas, comenzando el identificador por minúscula -a excepción de los identificadores de clases, que comenzarán por mayúscula también-):

```
double calculaTasaInternaDeRetorno()
{
        // aquí viene el código
}
```

en lugar del viejo estilo C de la forma calcula_tasa_interna_de_retorno. La verdad es que el más compacto estilo Smalltalk se está imponiendo en C++. Pero, bueno, cada uno a lo suyo: ¡si hasta existen literatos que trabajan con máquinas de escribir!.

En teoría la longitud de un identificador en C++ no tiene límites, aunque buena parte de los compiladores e intérpretes restringe dicha longitud a 32 caracteres como máximo. En orden a evitar problemas de portabilidad es aconsejable el mantenimiento de dicho límite, aunque a los efectos pedagógicos de este libro tal restricción no será contemplada con demasiada fruicción.

Es aconsejable que la legibilidad del código sea reforzada con la introducción (con las única restricción de la composición tipográfica) de espacios tras las comas, entre los identificadores y los paréntesis, y entre los operadores y los identificadores.

Una última cuestión: es frecuente que en porciones cortas de código el desarrollador opte, en una curiosa furia economizadora, por significar en la misma línea la cabecera y el bloque de una determinada estructura, como por ejemplo:

```
if ( condicion ) hazAlgo();
```

en lugar de disponerlo de la más correcta forma:

```
if ( condicion )
        hazAlgo();
```

¿la diferencia? Por un lado, con esta última sintaxis se es coherente con cualquiera de los estilos de indentación empleados (el cuerpo siempre se indenta un nivel); por otro lado, a la hora de chequear el código con un

debugger, de esta forma podremos establecer un *punto de ruptura* en una línea u otra, fraccionando más el análisis y, por tanto, facilitando la detección de posibles errores.

Bien: esto es todo. Dumas solía decir: "*Prefiero los malvados a los imbéciles; aquéllos, por lo menos, descansan*". Y es que pocas cosas hay tan insoportables como la gratuita e inmisericorde pesadez. Sepa de cualquier manera el lector que existen textos enteros consagrados a esta materia (como el de Indian Hill para C), así como manuales de corporaciones con sus correspondientes normativas estilísticas "de empresa". Vayamos, por fin, a C++ y a *las clases*.

PRIMER ACERCAMIENTO A LAS CLASES

Recordemos, retomando los escuetos conceptos apuntados en el capítulo 2, que una *clase* en C++ equivale a un tipo definido-por-el-usuario. No se trata aquí de una especial "estructura de datos" representando una combinación determinada de tipos predefinidos, ni tampoco de una particular ligazón de determinadas funciones con ciertos structs (como en C), sino de una auténtica abstracción encapsuladora tanto de los datos como de los métodos intrínsecamente ligados a éstos. Establezcamos, en un primer intento de aproximación formal al concepto, con un ojo crítico básicamente dolido de pedagogía, los siguientes ejemplos, que pasaremos a comentar línea a línea:

```
class ClaseVacia {                                                    // 1
};      //atención al punto y coma
        // tras la declaración de la clase

class ClaseSinMetodos {             // sólo datos                    // 2
                                    // equivale a un 'struct' de C
public:                                                               // 3
        long numeroDNI;
        char letraDNI;
private:                                                              // 4
        char* nombreCliente;
};

class ClaseSinDatos {                                                 // 5
        char* miClaveDeAcceso();//acceso PRIVADO por defecto          // 6
public:                             //cambio a acceso PÚBLICO
        int suma( int a, int b) { return a + b; }                     // 7
private:                            //acceso PRIVADO de nuevo
        long edadDeLolaFlores;
};

class ClaseConDatosYMetodos {                                         // 8
private:
        long numeroDNI;
        char letraDNI;
public:
        char calculaLetraDNI( long );
```

```
};

char ClaseConDatosYMetodos::calculaLetraDNI( long numero )                    // 9
{
        // función miembro definida fuera del ámbito de
        //  descripción de su clase y que accede a éste
        //  mediante el operador :: cualificador de ámbito.
        static char letra[] = "TRWAGMYFPDXBNJZSQVHLCKE";
        return letra[ numero & 23 ];
}
```

1. En primer lugar observamos una "clase vacía". De la misma forma que en la teoría matemática de conjuntos un conjunto vacío es distinto de un conjunto conteniendo el elemento cero o aún de un conjunto conteniendo al conjunto vacío, en C++ las instanciaciones (objetos) de una "clase vacía" cumplen que

```
ClaseVacia objetoDeClaseVacia, otroObjetoDeClaseVacia;
if ( sizeof( objetoDeClaseVacia ) != 0 )
        cout << "SIEMPRE se cumple\n";
if ( &objetoDeClaseVacia != &otroObjetoDeClaseVacia )
        cout << "SIEMPRE se cumple\n";
class ClaseNoVacia {
        ClaseVacia* miPunteroAObjetoDeClaseVacia;
} objetoDeClaseNoVacia;
if (    sizeof( objetoDeClaseNoVacia )
        != sizeof( objetoDeClaseVacia ) )
                cout << "SIEMPRE se cumple\n";
```

2. Una clase que no contenga métodos posee una equivalencia funcional similar a la de un struct en C, a excepción de la cualificación del acceso a los miembros (en este caso datos) de la clase. Esto significa, sin más, que no podemos acceder directamente a lo que no esté expresamente declarado como público. Tenemos, así, informalmente, que:

```
ClaseSinMetodos miObjetoDeDatos;
miObjetoDeDatos.numeroDNI = 21428748;       // OK
miObjetoDeDatos.letraDNI = 'Q';             // OK
miObjetoDeDatos.nombreCliente = "Landrú"    // ERROR: el acceso a
                                                    // nombreCliente es PRIVATE
```

3. La etiqueta *public* otorga la calificación de acceso "público" a los miembros de una clase comprendidos entre aquélla y la siguiente etiqueta de calificación de acceso o bien el fin de la declaración de la propia clase. Adelantaremos, extendiendo el ejemplo del punto anterior, que tanto esta calificación como la comentada en el punto 4 (*private*) están ligadas a la capacidad de acceso a los miembros de una clase tanto desde el protocolo de descripción de otra clase como desde un objeto de la misma u otras clases.

4. La etiqueta *private* supone una calificación restrictiva de acceso con respecto a la etiqueta *public*, reforzando el concepto de ocultación de la información tan enfatizado por la OOP. Como veremos más adelante y repitiendo lo anotado en el punto 2, un objeto no puede acceder directamente a los miembros *private* de su clase:

```
// ...
claseSinMetodos objetoSinMetodos;
cout << objetoSinMetodos.numeroDNI;                    // Ok: legal;
cout << objetoSinMetodos.nombreCliente;        // ERROR: acceso a
                                                       // miembro PRIVATE
```

5. La clase sin datos constituye la primera gran diferencia formal con respecto a C: un objeto de una de tales clases posee únicamente servicios. No debemos asimilar tal clase como una mera estructura contenedora o aglutinadora de funciones que habrían de ser compartidas por los objetos de tal clase (que sí se ajustaría a una clase conteniendo únicamente funciones *static*). Realmente cada objeto utiliza cada método particularizado a través de un nivel adicional de indirección, cual es el puntero implícito **this**, al que dedicaremos un detallado comentario.

6. En una clase el acceso es *private* por defecto. Tal circunstancia puede ser modificada por cualquier otra etiqueta expresa de cualificación de acceso: *public* o *protected* (un tipo especial de restricción de acceso que estudiaremos cuando veamos la derivación de clases, y que por ahora asimilaremos como que surte los mismos efectos que *private*), que pueden aparecer sin restricción de cantidad en cualquier sección del código de descripción de la clase.

7. La función suma(int,int) se ha definido dentro del cuerpo de descripción de la clase. Esto supone que, automáticamente, tal función será considerada **inline** y, por tanto, convenientemente macro-expandida (observando, naturalmente, las restricciones que a tal respecto pueda imponer el compilador, según establecimos en el capítulo anterior).

8. Una clase con métodos y datos supone el modelo más general, del que los casos anteriores se constituyen en particularizaciones sólo prácticas a efectos didácticos, pero sin ningún valor formal (no existen por tanto, ahora lo podemos decir, esos "tipos especiales" de clases, aunque sí es posible que se den clases sin datos, etc. O sea, existen mujeres fatales, pero esto no quiere decir que las mujeres se dividan morfológicamente en normales y fatales, aunque otra cosa pensara Jardiel). Pensemos, así pues, en las clases como colecciones de atributos y servicios comunes a bien-definidos conjuntos arbitrarios de objetos, de forma que, en un nivel extensivo, podrían también verse como unas particulares colecciones conceptuales de objetos. Recordemos que estamos hablando de *tipos de datos* sin límite cualitativo virtual, por lo que podrían aplicarse a cualquier conjunto que admitiera una definición comprehensiva. Tomemos, por ejemplo, un conjunto arbitrario de personas asistiendo a un concierto. El subconjunto de tales espectadores que portara una camisa blanca podría ser identificado y

encapsulado en la clase EspectadorMedioConCamisaBlanca, de tal forma que cada persona con estas características sería un objeto de tal clase, independientemente del número de personas que en un determinado momento se ajusten a tal descripción.

9. La función calculaLetraDNI(long), declarada en el ámbito de descripción de la clase, se ha definido fuera del cuerpo de ella. Normalmente esto se realiza así para mantener, en lo posible, una límpida separación entre el interfaz o descripción de la clase y la implementación o definición de ésta, a la vez que para ocultar al observador exterior los detalles de esta última, pues serían distribuidos como módulos objeto. Definir la función de esta forma conlleva dos inmediatos efectos: por un lado ya no se aplicará a ésta la calificación automática de *inline* (aunque sigue siendo posible declararla como *inline* añadiéndole el prefijo correspondiente "a mano"), así como, por otro, se hará necesario un mecanismo de identificación y resolución de ámbito que permita a tal función gozar de exactamente la misma operatividad que hubiera tenido si definida en el cuerpo de la clase, lo que será implementado mediante el uso del operador cualificador de ámbito (**::**), que adscribirá el código de la definición a una clase determinada (en este caso, a ClaseConDatosYMetodos). Ojo: el operador **::** permite realizar esto únicamente con funciones miembros, de forma que el compilador flagelaría como error el siguiente código, por el que se intenta realizar una asignación a una variable usando tal operador:

```
ClaseSinMetodos::numeroDNI = 21435019;          // ERROR
```

y la razón es bien clara: las funciones miembros son "servicios" conceptualmente iguales para todos los objetos de la clase, obviando el hecho de que operen con datos distintos, mientras que la construcción anterior sugiere que estamos asignando un valor de una variable a la clase en sí, pero la clase no es un objeto: es sólo un ente abstracto. Quizá lo veamos más claro en el siguiente ejemplo:

```
class Persona {
public:
        char sexo;              // 'M' por masculino, 'F' por femenino
        // resto de la descripción de la clase
};
// la siguiente asignación es errónea, porque sugiere que
// el tipo abstracto 'Persona' posee siempre sexo masculino,
// lo cual es mentira, a pesar de lo que puedan pensar algunos.
Persona::sexo='M';              // ERROR
// la próxima asignación es correcta, puesto que establece que
// el objeto Luis de tipo Persona es de sexo masculino.
Persona Luis;
Luis.sexo = 'M';
```

Puede darse el caso, sin embargo, que deseemos que todos los objetos compartan una misma variable o constante: entonces deberíamos usar un miembro estático (*static*) común a todos ellos y cuya noción, como es costumbre, desarrollaremos más adelante.

Debemos recordar que estamos trabajando con un lenguaje de jerarquía dual: las clases son distintas de los objetos, a diferencia de lo que, por ejemplo, ocurre en el lenguaje SELF.

Bien, superado este fugaz acercamiento, repasaremos seguidamente la estructura y características formales de las clases.

DEFINICIÓN DE CLASE Y MIEMBROS

De acuerdo a lo anticipado en el epígrafe anterior, las clases pueden particionarse en cuerpo y cabecera. La cabecera asocia a éstas una etiqueta o *especificador de tipo* (el nombre de la clase), a la vez que explicita, como más adelante veremos, las características adheridas a las mismas mediante el mecanismo de derivación de clases (circunstancia que obviaremos en este momento), resultando en la siguiente codificación general:

```
class Cliente {
        // cuerpo de descripción de la clase
};
Cliente fulano, mengano, zutano;        // definición de objetos
                                                    // de tipo "Cliente"
```

que equivale a:

```
class Cliente { /* descripción */ } fulano, mengano, zutano;
```

El cuerpo de la descripción de una clase -esto es, su definición- consta, por otro lado, de los siguientes elementos formales, encerrados entre llaves (**{};**):

Notación C++	Notación OOP
cualificaciones de acceso	exportabilidad de atributos
variables de datos miembros	variables de instanciación
declaraciones de funciones miembros	mensajes
definiciones de funciones miembros	métodos

Las funciones y variables (entre las que se incluyen punteros y referencias a otros objetos de la misma u otras clases) se denominan, respectivamente, **funciones miembros** y **datos miembros** de la clase a la que pertenecen. Veámoslo en detalle.

DATOS MIEMBROS DE UNA CLASE

Los **datos miembros** se declaran, dentro del protocolo de descripción de la clase, de la misma forma en que en C se declaran las variables: exactamente igual que si las estuviéramos declarando en un típico *struct*. Recordemos de nuevo que una clase equivale a un nuevo tipo definido-por-el-usuario, de forma que al igual que declaramos variables y punteros de tipos predefinidos (como *char* e *int*), también podemos declarar objetos, punteros y referencias a objetos de otras clases:

```
class Familia {
        char* domicilioHabitual;
        long numeroDePolicia, distritoPostal;
        char* ciudad;
        Persona cabezaDeFamilia;
        Persona conyugeDelCabezaDeFamilia;
        Persona* hijos;
        // etc., etc.
};
```

Vemos, así, que se han declarado variables de tipo *long* y punteros a *char*, junto con variables de tipo *Persona* y punteros a objetos de tipo *Persona*, que muy bien pudieran corresponderse a la clase ejemplificada poco antes.

Para declarar un objeto de una clase como miembro de otra clase es necesario que aquélla haya sido definida. O sea, en nuestro caso: la declaración de objetos del tipo *Persona* en la clase Familia hace necesario que la clase Persona haya sido anteriormente definida. Esta restricción se alivia si lo que se declaran, en vez de objetos, son punteros o referencias a otra u otras clases: en estos casos basta con que se anteceda una declaración de la clase en cuestión, pudiéndose definir más adelante en el código. Veámoslo en un simple ejemplo:

```
class Politico;                                    // declaración de clase
class ComitePara laSalvaguardaNacional // declaración de clase
{                                                  // comienza la definición
        // definición de clase
        Politico* consejoDeDireccionCorrompido
        Politico* directorFiguranteDePaja;
        Politico& samaniego, iriarte;
        // seguidamente se declara (se "define") como miembro
        //  de esta clase un "objeto" de una clase todavía
        //  no definida (no terminada de definir),
        //  cual es la presente clase
        Politico hombreHonesto;                    // ERROR
        // es correcta, sin embargo, la declaración de un puntero
        //  a la misma clase que se está definiendo, pues ésta
        //  "ya se ha declarado", como vemos a continuación.
        ComiteParaLaSalvaguardaNacional*
                comiteDeNuestroPaisVecino;
        // …
};      // aquí termina la definición de la clase
// más adelante en el código
class Politico {
        // definición de la clase Politico
};
class EspeciesAutoLucrativas {
                // sigue la definición de la clase
        // ahora sí se permite la inclusión como miembro
        //  de esta clase de un objeto de la clase "Político",
        //  pues esta clase ya ha sido definida
        Politico cargoXDeLaAdministracion;         // OK
        // …
};
```

Rizando el rizo -a pesar de que en la práctica a veces es inevitable-, hemos declarado un puntero incluso a la misma clase que se estaba definiendo. ¿Es esto correcto? ¡Efectivamente! Se cumplen las condiciones previstas: la declaración de la clase es anterior a la declaración del puntero, y se define (se termina de definir) más adelante.

De todas formas el lector se estará preguntando a estas alturas: "¿pero de dónde sacará el autor estos ejemplos tan ridículos? ¡Esto no tiene pies ni cabeza!". Bien, la verdad es que estas exposiciones me resultan bastante más costosas que si de "materias prácticas" se tratara: matrices, vectores, listas enlazadas, cadenas, árboles binarios, etc. constituyen un panorama ampliamente desarrollado en papel impreso: existen ejemplos, contraejemplos y recontraejemplos. Pero ocurre algo sorprendente: la experiencia me ha enseñado que el novicio en C++, sobre todo el que proviene de otros lenguajes clásicos de programación, rápidamente ve en estas estructuras una curiosa forma de desarrollar su viejo y conocido esquema funcional, por lo que, en un abrir y cerrar de ojos, se olvida de las consideraciones teóricas que subyacen bajo los conceptos de clase y objetos y se limita a modificar levemente su esquema funcional. En definitiva: se pierde más de lo que se gana. Indudablemente el lector con experiencia podría recuperar más adelante tales conceptos sin demasiados problemas, pero esto parece más adecuado para seminarios intensivos o cursos de especialización. En lo que a esta introducción al lenguaje C++ se refiere, tal y como se estableció en el capítulo 1, el enfoque será siempre el que proporciona la OOP, ayudándonos, si es posible, de comparaciones antropomórficas, para que el lector pueda "agarrar" esos conceptos, inicialmente tan difusos, de *encapsulación*, *abstracción*, etc. De cualquier forma tendremos ocasión de revisar, al menos, la clase *string* y algunas otras de interés. Claro que esto no es como una guia de una guia del *Ulysses* de Joyce, así que retomemos el hilo.

OCULTACIÓN DE LOS DATOS

Hemos visto que los *datos miembros* se han declarado indistintamente como *públicos* o *privados* en las clases anteriores. En realidad la OOP enfatiza una propiedad que se denomina **ocultación de la información** y que, básicamente, consiste en que los datos no podrán ser accedidos directamente, sino que tal acceso será realizado a través de funciones específicas. ¿Qué se pretende con esto? Pues que el usuario (por desarrollador) no tenga acceso a la representación interna de la clase. Imaginemos una clase en la que los datos internos están dispuestos en ficheros secuenciales. Si se pudiera acceder sin restricción a estos datos directamente, un programador podría generar código basado en la disposición física secuencial, de forma que si en un momento dado decidiéramos cambiar la disposición interna de los datos a una lista enlazada, aqúel código debería ser totalmente re-escrito. ¿Solución? Mantener los datos como *privados* e implementar una o varias funciones *públicas* de acceso a los datos, de manera que el programador "ignorará" la disposición interna de los datos y manejará únicamente funciones del tipo buscaClaveCliente(long) y listaClientesEntreCodigos(long, long). Si con tal disposición deseáramos cambiar la representación interna de una lista enlazada a una base de datos relacional, efectuaríamos los pertinentes cambios en la definición de la clase y, por supuesto, en las funciones de acceso, pero el código ya escrito (que usaría únicamente un prototipo inalterable de función miembro) no tendría que ser retocado. Bien: ésta es la idea. Para llevarla a buen fin C++ proporciona un sistema de cualificación de acceso finamente granulado a través de etiquetas, como ya hemos podido ver, y que consiste en las siguientes:

> **public**: todo miembro de una clase (función o variable) que se encuentre bajo esta cualificación de acceso podrá ser accedido sin restricción desde cualquier punto del programa.
>
> **private**: un miembro de una clase bajo esta etiqueta únicamente podrá ser accedido por las funciones miembros. También podrá ser accedido por un tipo especial de funciones conocidas como *friends* (amigas) y que veremos más adelante.
>
> **protected**: este tipo de miembros tendrán la consideración de *private* en lo que a su clase respecta con relación al programa. Serán, sin embargo, accesibles desde las clases *derivadas* de ésta, tal y como veremos más adelante.

Entonces, perguntará inquietado el lector, ¿para leer, modificar o imprimir cualquier variable "privada" habrá que implementar sendas funciones especiales adicionales? ¡En efecto! O sea, si tenemos

```
class Cliente {
private:
        char* nombre;
        // etc.
};
Cliente miClienteDePrueba;
```

para, por ejemplo, imprimir el nombre del cliente, **no** podríamos codificar

```
cout << miClienteDePrueba.nombre;     // error: acceso a
                                                    //  miembro private
```

sino que tendríamos que definir una función "especial" que realizara tal cometido. Así, podríamos por ejemplo declarar en la clase Cliente una función imprimerNombre() que realizara la tarea. Veámoslo:

```
class Cliente {
public:
        void imprimeNombre()
        {
                cout << nombre;
        }
private:
        char* nombre;
        // etc., etc.
};
```

De esta manera la impresión requerida se produciría mediante el siguiente código:

```
miClienteDePrueba.imprimeNombre();
```

Pero ¡bueno! -continuaría el lector-, esto quiere decir que aumentaremos el tamaño del código y disminuiremos sus prestaciones en tiempo de ejecución, pues lo que en C se realiza de forma directa aquí tiene que sobrellevar la penalización de la llamada a una o más funciones. De acuerdo: es cierto, pero sólo en una muy corta medida. El aumento del tamaño del código es despreciable en programas no triviales (superiores a mil líneas), mientras que la *performance* del código no tiene que verse disminuida si usamos del mecanismo de *inlining*. De hecho, como el avispado lector ya habrá observado, la función imprimeNombre() ha sido definida dentro del ámbito de descripción de la clase, por lo que, de acuerdo con lo ya visto, será automáticamente etiquetada como *inline*, de manera que, al efectuarse una macro-sustitución en cada llamada, no se producirá penalización alguna en *run-time*. La función en cuestión también podría haber sido declarada en la clase y definida fuera del ámbito de ésta (algo más acorde con la filosofía de separación del interfaz y la implementación, base de la OOP), pudiendo elegir entonces el desarrollador si etiquetarla o no como *inline*. La conveniencia o no de incurrir en la penalización antes anunciada es dejada, sin más, en manos del usuario del lenguaje.

Por defecto, si no se indica ninguna etiqueta, las clases poseen la cualificación de acceso *private*, reforzando así la idea de ocultación de la información. De hecho, la única diferencia en C++ de una *class* con respecto a un *struct* es que este último posee por defecto la cualificación de acceso *public*. ¡Vaya! Entonces, ¿un *struct* en C++ no es igual a un *struct* en C? No, naturalmente: obviando el tema de la cualíficación de acceso por defecto y si se explicita tal cualificación mediante etiquetas para cada miembro, donde quiera que aparezca el término *class* éste puede ser sustituido sin más por el de *struct*. Lo cierto es, sin embargo, que en la comunidad C++ se utiliza

siempre la *class*, reservándose el *struct* prácticamente para lo mismo que en C.

Consecuencia directa de este esquema de ocultación es la siguiente premisa: no se pueden añadir miembros (datos o funciones) a una clase una vez que ésta ha sido definida. ¿Por qué? Bien, resultaría absurdo que hubiéramos derrochado esfuerzos en desarrollar el acceso a la representación interna de una clase únicamente desde unas determinadas funciones públicas y que, por otro parte, cualquier desarrollador, con la simple adición de una función miembro a la clase ya definida pudiera obtener, sin más, tal privilegio de acceso (toda función miembro tiene acceso a los miembros *private* de la clase a que pertenece). ¿Que se pierde flexibilidad? Está claro. Pero, ¿que dirían ustedes si, por ejemplo, la ley permitiera a los herederos añadir cláusulas al testamento de una persona ya fallecida? ¿Que no es justo? Bueno, ésta es la perenne queja de los posibles herederos y, a la vez, la carga del testador. Existe en C++, empero, un mecanismo para flexibilizar y personalizar las clases sin que sea necesario modificar directamente el código ya escrito: la *derivación* de clases, materia de próximas entregas de esta introducción.

¿Existe alguna norma para la colocación de las etiquetas de acceso? Bien, en AT&T, por ejemplo, se suele comenzar la definición de la clase con la sección *public*, seguida de la *protected* y de la *private*. ¿Por qué? Pues porque parece que la exposición en primer lugar del interfaz público, con los métodos "de uso" de la clase, puede dar una idea más precisa del cometido abstracto de ésta, a la vez que explicita más rápidamente que de otra forma la necesidad y uso de las variables y funciones *private*. En otros textos, sin embargo, el orden es precisamente el inverso: se estima que el conocimiento inicial de la representación interna de la clase conllevará una más rápida comprensión de los mecanismos que la integran. En esto, como en tantas otras cosas, dirimirá la elección personal del lector. En mi caso me decanto por el primer estilo.

Una última nota: la *ocultación de la información* es una pilar teórico de la OOP y generalmente denota buen diseño, pero no es una característica de obligatoria aplicación y, por ende, es posible diseñar clases vulnerando tal regla. Pero para esto, como para ejercer la poligamia y el delito, hay que tener bastante práctica y encima uno siempre acaba cometiendo algún error, por lo que el lector deberá siempre intentar seguir el esquema de ocultación de datos, lo que por otro lado le proporcionará grandes satisfacciones en el futuro.

FUNCIONES MIEMBROS: VIEJAS SIERRAS CON DIENTES NUEVOS

El conjunto de las funciones miembros de una clase representa, básicamente, la expresa delimitación de las operaciones que un usuario puede efectuar sobre tal clase. Desde el punto de vista de la OOP, la suma de estas funciones sería calificada como el conjunto de posibles mensajes a los que los objetos de tal tipo (por clase) podrían responder. En la práctica, empero, tales funciones podrían ser vistas como una restricción del concepto general de función al ámbito de las clases. Veamos un primer ejemplo:

```
class Persona {
public:
        void imprimeDNI();              // función sin argumentos
        void imprimeNombre();
        long leeDNI();
        char* leeNombre();
        int calculaEdad( Fecha );       // argumento: un objeto
                                                    //  del tipo Fecha
        Ciudad& leeCiudad();            // devuelve una referencia a
                                                //  objeto del tipo Ciudad
        // …
private:
        long numeroDNI;
        Fecha* nacimiento;
        char* nombre;
        // …
};

inline void Persona::imprimeNombre()
{
        cout << nombre;
}
```

Vemos que la pura sintaxis funcional es como en ANSI C, con la diferencia que argumentos y tipos de retorno pueden ser, aparte de los predefinidos, los correspondientes a tipos definidos-por-el-usuario mediante clases, sin ninguna distinción visible entre unos y otros: ésta es la grandeza de C++.

¿Existen distintos tipos de funciones miembros? Bien, en realidad sí. Algunos autores basan la partición de estas funciones en grupos atendiendo principalmente a la finalidad de éstas. Así Stan Lippman, por ejemplo, las clasifica en funciones de gestión, de implementación, de ayuda y de acceso, aunque avisa que no existe diferencia física entre tales grupos, estableciéndose éstos con fines preminentemente didácticos, sin valor formal efectivo. En lo que a nosotros respecta consideraremos únicamente tres grupos: en primer lugar el formado por un tipo especial de funciones miembros llamadas *contructores* y *destructores*; seguidamente, el constituido por los *operadores*, bien de conversión bien por sobrecarga; en el último grupo, finalmente, se despacharían todas las demás funciones. Tendremos ocasión de revisar en más detalle tales grupos.

EL PUNTERO IMPLÍCITO "THIS"

Volvamos a sumergirnos en las clases y en la conceptualización dual clase-instanciación/tipo-objeto. Imaginemos que hemos dotado a una clase, convertida ya en un verdadero tipo definido-por-el-usuario, de una representación interna sustentada en variables (presumiblemente *private*s), de tal forma que esta "estructura" sería traspasada a cada uno de los objetos de tal tipo. Cada objeto, seguidamente, podría inicializar y "rellenar" de forma conveniente tal "estructura". O sea, un número 'n' de objetos supone un igual número 'n' de representación internas: cada objeto dispone de un área de almacenamiento propia -no compartida por los demás objetos en el caso más general- dedicada a soportar tal representación. ¿Qué ocurre, sin embargo, con las funciones miembros? ¿Cada objeto *repite*, como ocurre con las variables, todas las funciones definidas en la clase de que toma el tipo? ¡No, por supuesto! Repetir el código de la función para cada uno de los objetos constituiría una pérdida de espacio de muy difícil justificación. En realidad todos los objetos de una clase usan de la misma única copia de cada una de las funciones. Esto es: todos los objetos comparten el mismo código para cada función miembro. Pero, entonces, si suponemos que una determinada función miembro, mediante la general codificación en el ámbito de definición de la clase, tiene que acceder a la representación interna de un objeto, ¿cómo podrá discernir tal función, basándose en un mismo código común, el objeto con que deberá operar para tratar con sus variables privadas? Intentaré clarificar el problema con el siguiente ejemplo:

```
class Proveedor {
public:
        void imprimeNombre();
        void imprimeDireccion();
        void imprimeFichaProveedor();
        // …
private:
        char* nombre;
        char* direccion;
        // …
};

void Proveedor::imprimeNombre()
{
        cout << nombre << "\n";
}

void Proveedor::imprimeDireccion()
{
        cout << direccion << "\n";
}

void Proveedor::imprimeFichaProveedor()
{
        imprimeNombre();
```

```
        imprimeDireccion();
}
```

Veamos si el lector ha entendido la naturaleza del problema: es evidente que si declaramos un objeto como del tipo Proveedor y seguidamente le enviamos el mensaje imprimeNombre,

```
Proveedor proveedorDescargandoEnElMuelle;
proveedorDescargandoEnElMuelle.imprimeNombre();
```

la función miembro imprimeNombre() conocerá en este caso con exactitud a qué objeto ha de ser aplicada, evidentemente. Asi mismo, cuando codificamos

```
proveedorDescargandoEnElMuelle.imprimeFichaProveedor();
```

este nuevo mensaje o función miembro también sabrá que el objeto a que debe aplicarse, al igual que en el ejemplo anterior, es el proveedor-DescargandoEnElMuelle. Pero, ¿qué pasa con las funciones miembros que se encuentran, sin referencia directa alguna, en el cuerpo de esta última función? O sea, ¿a qué objeto se aplicarán los mensajes imprimeNombre() e imprimeDireccion(), contenidos en la función imprimeFichaProveedor(), toda vez que la sintaxis no es explícita como en los casos anteriores? Bien, en realidad todas las llamadas a miembros de una clase (tanto variables como fuinciones) desde el protocolo de descripción de ésta se direccionan a un puntero implícito, denominado *this*, que contiene la dirección del objeto de esa clase por medio del que se ha producido la llamada. Esto es, si explicitamos tal puntero, podríamos perfectamente escribir:

```
void Proveedor::imprimeNombre()
{
        cout << this->nombre << "\n";
}

void Proveedor::imprimeFichaProveedor()
{
        this->imprimeNombre();
        this->imprimeDireccion();
}
```

donde *this*, en este caso, habría sido ya implícitamente declarado de la siguiente forma:

```
Proveedor *const this;
```

De esta forma, en nuestro caso, ya podemos inferir que los mensajes del cuerpo de la función imprimeFichaProveedor() serán direccionados al objeto proveedorDescargandoEnElMuelle, como era intuitivo suponer. Naturalmente que el usuario puede explicitar el puntero *this* en los cuerpos de todas las funciones miembros, aunque esto, lejos de aclarar, oscurece más el código y, en la práctica, simplemente no se hace.

Hemos visto que *this* es un puntero constante a un objeto de la clase a que pertenece el miembro, de tal forma que *this* no puede ser cambiado, aunque sí puede serlo el objeto apuntado por él, o sea **this*. Podemos, pues, usar sin restricciones esta nueva característica, como por ejemplo en el valor de

retorno de las funciones miembros. Vamos, en base a esto, a revisar y reescribir la clase antes propuesta:

```
class Proveedor {
public:
        Proveedor imprimeNombre();
        Proveedor imprimeDireccion();
        void imprimeFichaProveedor();
        // …
private:
        char* nombre;
        char* direccion;
        // …
};

Proveedor Proveedor::imprimeNombre()
{
        cout << nombre << "\n";
        return *this;
}

Proveedor Proveedor::imprimeDireccion()
{
        cout << direccion << "\n";
        return *this;
}

void Proveedor::imprimeFichaProveedor()
{
        imprimeNombre().imprimeDireccion();
}
```

¿Qué hemos cambiado? Bien, hemos convertido imprimeNombre() e imprimeDireccion() en funciones con valor de retorno del tipo *Proveedor* y modificado convenientemente sus cuerpos. De esta manera si se produce el envío de uno de tales mensajes a un objeto determinado, como por ejemplo proveedorDescargandoEnElMuelle.imprimeNombre(), el resultado de tal envío será la impresión del nombre y el retorno del mismo objeto proveedorDescargandoEnElMuelle (**this*). Examinemos ahora la definición de la función imprimeFichaProveedor(). Hemos encadenado las dos funciones anteriores con el siguiente significado: si suponemos la llamada proveedorDescargandoEnElMuelle.imprimeFichaProveedor(), primero se ejecutará el bloque más hacia la izquierda, equivalente a this->imprimeNombre(), que en este caso equivale, a su vez, a proveedorDescargandoEnElMuelle.imprimeNombre(), que tras ejecutarse devolverá el objeto proveedorDescargandoEnElMuelle, al que seguidamente se aplicará el siguiente bloque a la izquierda, resultando en proveedorDescargandoEnElMuelle.imprimeDireccion(), que a su vez devolvería el mismo objeto al que podría aplicarse otro bloque, si lo hubiere.

FUNCIONES MIEMBROS CONSTANTES

Recordemos que, básicamente, un objeto consta de una representación interna y de un interfaz de métodos que acceden, manipulan y modifican tal representación. Pensemos, por otro lado, que la definición de una clase posibilita su uso en calidad de nuevo tipo definido-por-el-usuario, con sintaxis pareja a la de los tipos predefinidos. Una variable de un tipo incorporado puede declararse sin problemas como constante, pero ¿qué pasa con los objetos de una clase?

Si declaramos un objeto de una clase dada como constante e intentamos enviarle un mensaje en razón de una función miembro de tal clase que modifique la representación interna del objeto, el compilador señalará inmediatamente el error. En realidad a un tal objeto únicamente podrían serle aplicadas funciones miembros que mantuvieran invariante su estructura interna. Pero, ¿cómo ha de saber el compilador que tal o cual función es "segura" en este sentido? Pues merced a que el desarrollador habrá etiquetado tal función miembro como *constante*, tanto en su declaración como en su definición. Así, por ejemplo, podríamos reescribir alguna de las anteriores funciones de la siguiente forma:

```
class Proveedor{
public:
        Proveedor imprimeNombre() const;
        // ...
};

Proveedor Proveedor::imprimeNombre() const
{
        cout << nombre << "\n";
        return *this;
}
```

Por supuesto sólo podemos etiquetar como constantes las funciones miembros que no modifiquen la representación interna de los objetos (los datos miembros), obteniendo un error en compilación en caso contrario.

Si declaramos un objeto de una determinada clase como constante, a tal objeto sólo podrán serle dirigidos mensajes significados por funciones miembros constantes. Una función miembro constante podrá, sin embargo, ser llamada desde cualquier objeto, constante o no. O sea, que tenemos:

```
const Proveedor proveedorConstante;
Proveedor proveedorDeAccesorios;                    // ok
// seguidamente objeto constante llama a función constante
proveedorConstante.imprimeNombre();                 // ok
// objeto constante llama a función no const
proveedorConstante.imprimeDireccion();              // error
// ahora objeto no constante llama a función const
proveedorDeAccesorios.imprimeNombre();              // ok
```

En una función miembro declarada como constante, como por ejemplo la ya vista imprimeNombre(), el puntero *this* queda implícitamente declarado como

```
const Proveedor* const this;
```

Todo lo visto hasta ahora parece que viene en afirmar que una función miembro declarada constante no puede modificar un objeto, pero lo cierto es que **sí** puede. ¡Demonios!, pensará el lector: ¡Ahora sí que no se entiende nada!. Bueno, como tantas veces suele ocurrir en C++, el lenguaje proporciona fuertes mecanismos por defecto que sin embargo pueden ser totalmente obviados por el desarrollador: pensemos, por ejemplo, en el "fuerte chequeo de tipos", que puede ser totalmente anulado mediante la técnica llamada de "constructores virtuales". Bien, volviendo a las funciones *const*, debo decir que esta etiqueta se refiere a lo que se denomina "*constancia lógica*", en contraposición a la constancia física, indicando que tal característica es, en definitiva, una mera apariencia, que el usuario puede soslayar simplemente realizando un *cast* expreso de la siguiente forma:

```
void Proveedor::cambiaNombre( char* nombreNuevo ) const
{
        (Proveedor*)this->nombre = nombreNuevo;
}
```

Esto es, en C++ normalmente el deseo expreso del desarrollador prima sobre todo lo demás: algo que suscita odios y pasiones.

INTERFAZ E IMPLEMENTACIÓN

Como el lector ya habrá adivinado, las definiciones de las clases, constitutivas del interfaz del proyecto software, se registran en archivos de cabecera (de extensión .h, .hxx o .hpp), mientras que las definiciones de las funciones miembros, o implementación, serán encerradas en archivos con extensión .CPP ó .C. En el protocolo de descripción de las clases no deben caber definiciones, sino sólo declaraciones. Cuando más adelante expongamos ejemplos completos de análisis, diseño y codificación de clases comentaremos en detalle esta límpida separación.

6

ENTRADA Y SALIDA DE DATOS

Debido a la ansiedad (como relata Bertrand Meyer) que experimentan los novicios respecto de las operaciones de entrada y salida de datos, antes de seguir desarrollando las características generales de las "clases" vamos a practicar un sucinto repaso a la librería standard de i/o en C++.

INTRODUCCIÓN A LA LIBRERÍA "IOSTREAM.H"

La librería **iostream** es el único soporte estándar actual, según las especificaciones de AT&T, incorporado al lenguaje C++ para la gestión de las operaciones de entrada/salida de datos. Originariamente tal librería, implementada y diseñada por Bjarne Stroustrup, se denominó **stream**. A partir, sin embargo, de AT&T C++ 2.0 ésta fue reimplementada por Jerry Schwarz, con la significativa adición de los manipuladores y adoptando su actual nombre, constituyendo la base para el establecimiento de la librería i/o en ANSI C++ que, en todo caso, será una simplificación de la presente.

En esta librería, incluida en el archivo de cabecera *iostream.h*, se predefinen los siguientes cuatro objetos básicos, instancias de las clases ostream (salida: output stream) e istream (entrada: input stream).

- **cin** (leído *see-in*), que es un objeto predefinido de clase istream afectado a la entrada standard.

- **cout** (leído *see-out*), que es un objeto predefinido de clase ostream direccionado al dispositivo standard de salida.

- **cerr** y **clog**, objetos predefinidos de clase ostream para el tratamiento de salida mensajes de error, sin y con buffer respectivamente.

¡Un momento, un momento! ¿De dónde han salido estos objetos? ¿Quién, cómo han sido inicializados? ¿Y qué ocurre con ellos tras terminar la ejecución de nuestro programa? Bien, son buenas preguntas. ¿Las respuestas? Concisamente: tales objetos globales son declarados *static* en el

archivo de cabecera *iostream.h*, y resulta que las funciones *constructoras* de los objetos estáticos (locales o globales) son llamadas antes de la ejecución de *main()*, así como las *destructoras* son llamadas después de acabar *main()*. ¡Vaya! ¿Y por qué? Pues porque tal esquema permite, como en este caso, unas adecuadas y elegantes inicialización y destrucción de estructuras de datos en librerías, en su forma más general. ¿Y por qué así y no de otra manera? Porque las demás alternativas son peores o demasiado especializadas para cada librería. Y ya está bien de preguntar. Sigamos.

En las clases istream y ostream se sobrecargan[11], respectivamente, los operadores para la ejecución de las operaciones de *inserción* o entrada de datos (**<<**) y *extracción* (**>>**) o salida de datos. De esta forma la representación codificada de I/O podría ser:

```
cout << datos;          // inserción de datos en el
                                // objeto o "stream" cout
cin  >> datos;          // extracción de datos del
                                // objeto o "stream" cin
```

Veamos seguidamente el típico programa de saludo:

```
#include <iostream.h>
int main( int, char** ) {
        cout << "Hola, C++";
        return 0;
}
```

Notamos, pues, que la cadena "Hola, C++" se inserta en el objeto cout, representación virtual del dispositivo de salida (normalmente la pantalla del terminal).

Bueno, pero ¿por qué utilizar los operadores '<<' y '>>' en lugar de otros? ¿Y por qué, en todo caso, usar dos operadores distintos cuando la distinción en sí de las operaciones fácilmente se localiza en los objetos predefinidos? En primer lugar tenemos que no podemos "crear" nuevos operadores: tenemos que conformarnos con los existentes. Seguidamente, en efecto, cabría preguntarse: ¿cuál? ¿quizá el operador '='? Lo cierto es que las expresiones (sobre todo al encadenarse) pueden complicarse sobremanera, y una no muy intuitiva adecuación del operador a la operación dada podría dificultar, a veces de forma insuperable, la legibilidad del código. Y es tal inteligibilidad la que se pretende al elegir dos operadores asimétricos que por su forma ya sugieren un direccionamiento (de datos, de objetos, etc.). Realice el amable lector una sencilla prueba: tras terminar el presente

[11] Recuerde el lector lo ya dicho sobre sobrecarga de funciones (de las que los operadores son un caso especial): el mismo interfaz con distintas implementaciones contextuales. De esta manera los operadores normalmente usados para desplazamiento de bits son "redefinidos" para actuar como "insertores" y "extractores" en sus relaciones con *streams*.

capítulo cójase la tabla de operadores y sustituya los operadores '<<' y '>>' por los de su elección en las expresiones de este texto. Colija el lector, seguidamente, la adecuación o no de tal elección[12].

Cualquiera de los operadores puede ser utilizado en forma encadenada. Esto es,

```
cout << "El número " << 7 << " es cabalístico." << "\n";
// la siguiente línea es equivalente a la anterior
cout << "El número 7 es cabalístico\n";
```

debido a la definición del operador sobrecargado, que devuelve, como veremos, una referencia a ostream:

```
ostream& operator<<( tipoDeDato );
```

De esta forma, la operación de "*inserción*" de un dato en el objeto cout devuelve el mismo objeto, preparado para una posible nueva "*inserción*" de datos. El código anterior podría, pues, ser visto así:

```
( ( ( ( cout        << "El número" ) << 7 )
                    << "es cabalístico." )
                    << "\n"; )
```

La idea intuitiva del "funcionamiento" del objeto **cout**, por ejemplo, es que los datos se van insertando en un buffer interno que una vez lleno, o forzado por un manipulador o una función miembro, direcciona su volcado hacia un dispositivo de salida preestablecido, que por defecto (igual para entrada que para salida) es la pantalla del terminal, aunque, como es fácil suponer, tal circunstancia puede ser variada direccionando el volcado a otro dispositivo.

Es muy aconsejable el estudio por los lectores de los archivos de cabecera de las clases **stream** en su implementación C++, pues son éstas librerías muy chequeadas con una ortodoxa y cuidada implementación de la derivación de clases y en las que se da una inteligente y particionada (en el sentido matemático del término) profusión de funciones miembro.

[12] No olvidemos que C++ es un lenguaje que permite al desarrollador una gran flexibilidad: el lector disconforme con la elección de operadores podría, por ejemplo, derivar sus propias clases de *istream* y *ostream* añadiendo a éstas, seguidamente y tras lo que puede ser un arduo trabajo, un operador distinto que, caracterizado como *inline*, sustituyera a aquéllos, creando a la vez unos nuevos *streams* **micout** y **micin**. Tal posibilidad existe, como también la de automutilarnos, pero esto no dice nada de la conveniencia de su aplicación.

MENSAJES DE I/O DIRIGIDOS A OBJETOS "*STREAMS*"

Hemos visto que la simple codificación:

```
cout << 8;
cout << "hola";
```

origina que se impriman sucesivamente en la consola un entero y una cadena. Pero, ¿cómo demonios sabe el objeto cout que el primero es un entero y el segundo no? O sea, ¿que pasa con la conocida necesidad de indicar el tipo del "dato" a imprimir?, algo tan habitual en 'C' como lo siguiente:

```
printf ( "He aquí un entero: %d \n", 8 );
```

La respuesta, a estas alturas, ya debe haberla adivinado el lector: los operadores de *inserción* y *extracción* son sobrecargados en las clases *istream* y *ostream*, de forma que el mismo operador (esto es, el mismo identificador de función, sea '<<' ó '>>') admite distintos argumentos, produciéndose en cada caso una llamada a la función específica con argumento(s) del mismo tipo que el objeto que se pretende *insertar* o *extraer* de un *stream*.

Examinándolo desde una óptica Orientada-a-Objetos, esta situación sugiere la siguiente interpretación: un objeto determinado (un entero, una cadena, etc.) dirige un mensaje de *extracción* (>>) o *inserción* (<<) al objeto *stream*, el cual, a su vez, procede a la siguiente evaluación general:

ORIENTADA-A-OBJETOS	FUNCIONAL
¿qué objeto envía el mensaje?	¿cuál es el tipo del objeto que envía el mensaje?
¿existe un método de respuesta para tal mensaje?	¿se ha definido en la clase a que pertenece el *stream* una función con un argumento del mismo tipo que el objeto que envía el mensaje? En caso que no: ¿existe una función miembro con un argumento de un tipo al que el tipo del objeto que envía el mensaje pueda convertirse por la aplicación de las reglas de resolución en sobrecargas de funciones?
El objeto *stream* envía un mensaje de respuesta.	Se produce una llamada a la función resultante de la evaluación anterior.

Llegamos, así, a la facilidad de que sea el sistema el que se ocupe del trabajo sucio, preocupándose de discernir la respuesta a aplicar en cada momento en razón de las características concretas del estímulo. De esta manera, por ejemplo, si declaramos una variable del tipo *int* y usamos de ella en expresiones con *streams*, el cambio del tipo de la variable a, verbigracia, *char* básicamente no nos habrá de afectar.

Vemos, pues, que los objetos predefinidos pueden manejar un elevado rango de tipos, pues en definitiva están provistos de funciones miembro que implementan métodos de respuesta a los mensajes (singularmente << y >>) que les envían objetos de distintos tipos (int, double, char*, etc.). Esto obedece a una codificación de la siguiente catadura:

```
class ostream : public virtual ios {
public:
        ostream& operator<<( const char* );
        ostream& operator<<( char );
        ostream& operator<<( short i ) {
                return *this << int(i);         // inline
        }
        ostream& operator<<( double );
        // etc., etc.
};
```

Tal versatilidad puede generar alguna confusión como, por ejemplo, en el caso de los punteros a char, pues este tipo de dato será asumido por el objeto **iostream** como una cadena tipo C, así que si quisiéramos usar la dirección contenida en tal puntero tendríamos que realizar un **cast** explícito de la forma

```
char* punteroAChar = "Prueba objeto cout";
cout << punteroAChar;                   // cadena apuntada por puntero
cout << ( void* )punteroAChar;          // dirección puntero
```

¡Ahora lo entiendo! -podría suspirar aquí el paciente lector-: entonces si, por ejemplo, deseara imprimir un objeto de un tipo *definido-por-el-usuario* a través de una clase, lo único que tendría que hacer es descomponer el objeto en datos que serían insertados individualmente en el objeto cout. ¿O no es así? Bien, básicamente sí es así. De esta manera podríamos codificar:

```
class Recluta {
public:
        void imprimeDatos()
        {       // función inline
                cout << "Datos recluta Ejercito de Tierra:\n";
                cout << numeroDNI << endl;
                cout << apodo << "\n";
        }
private:
        long numeroDNI;
```

```
        char* apodo;
};
```

siendo así que si definimos e inicializamos un objeto del tipo *Recluta*, como por ejemplo *miRecluta*, para imprimir los datos internos deberíamos codificar

```
miRecluta.imprimeDatos();
```

Entonces, ¿esto es correcto? ¿es así como se imprimen los objetos instanciaciones de clases?. Bien, la verdad es que no. Estamos tan acostumbrados a pensar siguiendo un esquema funcional que enseguida queremos componer funciones "al antiguo estilo" con todo lo que tenemos a nuestro alcance. Hay que cuestionarse lo siguiente: realmente C++ hace un gran esfuerzo para equiparar los tipos predefinidos con los *definidos-por-el--usuario* a través de la clase, mediante parecida sintaxis de declaración y definición, encapsulando las operaciones, etc., y sin embargo nosotros obviamos esta evidencia y decimos: "bien, los objetos de tipo predefinido con los operadores de *inserción* y *extracción*, mientras que las instanciaciones de clases con *funciones* específicas". No hay razón para establecer tal distinción: en efecto, la impresión del objeto *miRecluta*, de tipo *Recluta*, debería ser codificada así:

```
cout << miRecluta;          // !!!!!
```

Pero -dirán ustedes-, se supone que la clase *ostream* no posee una función miembro con argumento del tipo *Recluta*, pues el implementador de tal clase no sabía -ni afortunadamente sabrá nunca- nada sobre esta nuestra clase. ¿Qué pasa aquí? Bien, es muy sencillo: únicamente tenemos que sobrecargar, en la clase *Recluta*, los operadores de *inserción* y *extracción*, para que respondan a argumentos del tipo *Recluta*. ¿Qué cómo se hace esto? Más adelante lo veremos.

De hecho se encuentran con frecuencia en los textos de introducción a C++ ejemplos de sobrecarga de los operadores de inserción y extracción en el ámbito de una clase, pues tales se han asimilado fuertemente a las operaciones de entrada y salida. Así, verbigracia, estos operadores suelen sobrecargarse como indicadores de las operaciones de archivo y recuperación de objetos almacenados en "objetos ficheros", etc.

Vemos, en definitiva, que la flexibilidad en la aplicación de las operaciones de *inserción* y *extracción* es poco menos que ilimitada.

FUNCIONES MIEMBROS DE LOS *STREAMS*

Dadas las características de los **streams**, podría ser aconsejable la inclusión en un archivo de cabecera de constantes simbólicas para los items más usados, verbigracia:

```
const char campana =        '\007';
const char* flecha =        " ===> ";
```

pudiendo ser usados de la siguiente forma:

```
cout << "La respuesta es " << flecha << "NO" << campana << endl;
```

donde *endl* es una operación predefinida en *ostream* que inserta una nueva línea y, a la vez, vacía el buffer. A este último y único fin también podrían utilizarse las siguientes expresiones:

```
// fuerza el volcado del buffer
cout << flush;      // mediante un "manipulador"
cin.flush();        // mediante una función miembro
```

En realidad los streams predefinidos poseen gran cantidad de funciones miembros para el manejo de datos, entre las que cabría destacar:

cout

```
// inserta carácter en cout
cout.put( char caracter );
// inserta desde cadena[0] hasta cadena[longitud] en cout
cout.write( const char* cadena, int longitud );
// vacía el buffer en el dispositivo de salida
cout.flush();
// …
```

cin

```
// extrae un carácter del objeto cin
cin.get( char& caracter );
// iterador extractor que devuelve un caracter
cin.get();
// extrae línea desde buffer[0] a buffer[limite]
//  o hasta encontrarse con el delimitador indicado
cin.getline(        char* buffer, int limite,
                            char delimitador = '\n' );
// extrae una cadena desde direccion[0]
//  hasta direccion[longitud]
cin.read( char* direccion, int longitud );
// devuelve número de caracteres leídos
//  en el último cin.read( ... )
cin.gcount();
// devuelve a cin un carácter extraído
cin.putback( char caracter );
// desecha la extracción del buffer
//  desde buffer[limite] hasta el delimitador
cin.ignore( int limite, int delimitador = EOF );
// devuelve el siguiente carácter sin extraerlo
cin.peek();
// …
```

MANIPULADORES

En el apartado anterior hemos visto que se puede forzar el volcado del buffer de un *stream* bien mediante una función miembro bien a través de un manipulador. ¿Qué es, sin embargo, un *manipulador*? En pocas palabras: es un objeto (un objeto "función", por así decirlo) que cada vez que se utiliza equivale a una llamada a una determinada función miembro. Así, como ya hemos visto,

```
cout << flush;
```

equivale a

```
cout.flush();
```

O sea, la inserción del objeto flush en el objeto cout equivale a (o provoca) la llamada de la función miembro flush() en este último.

Esta técnica, ideada por Andrew Koenig, permite una sintaxis más clara y una mayor flexibilidad, pues las operaciones de entrada y salida no tienen que ser interrumpidas para la aplicación de determinadas funciones.

Los *manipuladores* con argumentos (?!) están definidos en el archivo "**ioma-nip.h**" y son:

```
setw( int )                     // establece la longitud del
                                //  buffer del campo de datos
setfill( int )          // establece el carácter de
                                //  relleno de campos de datos
setbase( int )          // establece la base numérica
                                //  de conversión
setiosflags( long )  // establece las marcas en el
                                //  vector de bits de control
resetiosflags( long)    // limpia todas las marcas
                                //  del vector de bits de control
setprecision( int )  // establece los digitos de precisión
```

La lista, por otro lado, de *manipuladores* sin argumentos está compuesta por:

```
oct             // determina base numérica octal
dec             // determina base numérica decimal
hex             // determina base numérica hexadecimal
flush           // fuerza el volcado del buffer
endl            // equivale a la aplicación
                //  consecutiva de '\n' y flush
ends            // equivale a la adición de '\n'
                //  y la aplicación de flush
ws              // se "come" (u obvia) un espacio en blanco
```

Lo cierto es que al lector no le habrá quedado muy clara la técnica en que se base el funcionamiento de los *manipuladores*, pero una explicación más detallada sobrepasaría los límites de esta introducción. Bástele saber que los *manipuladores* son extensibles (particularizables) y también que el desarrollador puede crear sus propios manipuladores. A estas alturas es una cuestión de fe.

FORMATO DE LOS DATOS DE ENTRADA/SALIDA

Hemos visto que ciertos manipuladores (como por ejemplo hex y setprecision(int)) pueden afectar el formato de las operaciones de entrada/salida. De hecho los manipuladores sin argumentos se definen en la clase *ios*, que es una clase base (superclase) de las clases *istream* y *ostream*. Esta clase dispone de distintas funciones miembros que pueden ser aplicadas a los *stream* predefinidos. ¡Alto aquí! ¿Un objeto va a usar funciones miembros de otra clase como si fueran suyas? ¡Efectivamente! Mediante la derivación de clases, los objetos de las clases derivadas públicamente de una clase base (cual es *ios* en este caso) pueden llamar directamente a las funciones miembros declaradas en la sección *public* de tal clase base. Bien, veamos algunas de tales funciones, aplicables tanto a objetos *ostream* como *istream*:

```
cout.fill( char );      // establece el carácter de relleno
                                    //  ( por defecto espacio)
cout.fill();                    // devuelve el carácter de relleno
cout.width( int );      // establece la longitud del
                                    //  campo de salida
cout.width();                   // devuelve la longitud del
                                    //  campo de salida
cout.precision( int );          // establece la precisión en decimales
cout.precision();               // devuelve la precisión.
cin.eof();                      // devuelve la condición booleana
                                    //  de fin de fichero
// ...
```

Los objetos *streams* mantienen, adicionalmente, un *estado interno de formato* para controlar las operaciones de formato a través de señales (*flags*) específicas. Tales señales pueden ser activadas o desactivadas mediante las funciones miembros de la clase *ios* setf(...) y unsetf(...), respectivamente. El establecimiento de un formato de salida podría ser codificado, en general, de la siguiente forma:

```
cout.setf( ios::FLAG );
```

donde FLAG habrá de ser sustituido por uno de los siguientes identificadores:

```
left                    // justificación a la izquierda
right                   // justificación a la derecha
internal        // justificación entre signo y número
dec                     // salida numérica en base decimal
hex                     // salida numérica en base hexadecimal
oct                     // salida numérica en base octal
fixed                   // salida de números con coma flotante
                        //  en notación regular
scientific      // números con coma flotante
```

```
                        //  en notación científica
showpos         // antecede un signo '+'
                        //  a los números positivos
showpoint               //  muestra el punto decimal y los ceros
                        //   tras éste en números decimales
uppercase               // salida de caracteres en mayúsculas
skipws          // salta espacio en blanco en entrada de datos
```

Existe, a la vez, la siguiente sobrecarga de la función setf (long):

```
long setf(      long flagOCampoDeBitsDeFormato,
                long campoDeBitsDeFormato );
```

donde el primer argumento puede ser una *flag* de las antes detalladas o un *campo* de los siguientes,

```
basefield               // base numérica
floatfield              // notación de números con coma flotante
```

de manera que cada campo (*Format Bit Field*) se aplica a unas determinadas señales (*Format Flags*). Así

```
ios::basefield      se aplica a las flags ->    ios::hex,
                                                ios::dec,
                                                ios::oct
ios::floatfield     se aplica a las flags ->    ios:fixed,
                                                ios::scientific
```

Pero, ¡vaya! ¿para qué es necesaria realmente esta última sobrecarga? Bien, resulta que la codificación setf(FLAG) resulta en que se activa una determinada *format flag*, pero sin inicializar el estado del campo a que se adscribe. Esto es, si escribimos

```
int numero = 10;
cout.setf( ios::hex );      // establece 'basefield'
                                //  como hexadecimal
cout << numero;             // salida: A
cout.setf( ios::oct) ;      // no resetea el campo 'basefield',
                                //  estableciendo dos bases distintas
cout << numero;             // salida: 10 !!! (la base no es octal)
```

tenemos que al aplicar dos distintas bases sobre el campo ios::basefield, éste vuelve a su valor por defecto: la base decimal. ¿Cómo se soluciona esto? Bien, la versión sobrecargada primero resetea el campo de formato a 0 y seguidamente establece en él el valor del *flag*. ¿Y si quiéramos conservar el antiguo valor del campo? Precisamente long setf(long, long) devuelve tal valor, por lo que podrá ser almacenado y recuperado posteriormente.

UN SENCILLO EJEMPLO PRÁCTICO

Veamos algunas líneas de código mostrando el manejo elemental -muy elemental- de la librería "**iostream.h**", haciendo uso de funciones miembros y de manipuladores:

```
#include <iostream.h>
int main( int, char** )
{
        // inserciones                  // salida a dispositivo estándar
                                                //  y comentarios
        cout << hex << 10;              // A
        cout << oct << 9;               // 11
        cout << oct << 9
                << hex << 10;           // 11 A
        cout << char( 67 ); // 'C' en un PC
        cout << 'C';                    // C
        cout << int( 'C' );    // 67 en un PC ( CHAR es distinto
                                                //  de INT en C++)
        cout.precision( 2 );  // establece la precisión SÓLO
                                                //  para la próxima inserción
        cout << 1.23234;                // 1.23
        cout << 1.23234;                // 1.23234
        cout.width( 5 );                // establece la longitud de
                                                //  la próxima inserción
        cout << 2;                              //     2
        cout << 2;                              // 2
        cout .width( 8 );               // establece la longitud SÓLO
                                                //  de la próxima inserción
        cout.fill( '0' );               // establece el carácter de
                                                //  relleno para completar "width"
        cout.precision( 3 );
        cout << 1.28317;                // 0001.283
        cout.width( 9 );
        cout << 1.28317;                // 001.28317
        cout.fill( ' ' );               // restablece el caracter de
                                                //  relleno a Blanco
        return 0;
}
```

Observamos que algunos métodos (como width(int) ó precision(int)) se aplican únicamente en la inserción de datos en el objeto **cout** inmediatamente posterior a su llamada, restaurándose, tras ésta, los valores anteriores. Otros, empero, (como fill(char)) cambian de forma duradera el comportamiento del objeto **cout**.

MANEJO DE FICHEROS

Las clases para el manejo de operaciones de entrada/salida en ficheros están contenidas en el archivo "**fstream.h**", el cual automáticamente incluirá -si no ha sido hecho ya- el archivo "iostream.h". Tales clases, a nuestros efectos, se reducen a las siguientes:

. *ifstream*: para ficheros que se abren en modo lectura

. *ofstream*: para ficheros que se abren en modo escritura
. *fstream*: para ficheros abiertos en modo lectura/escritura

Entonces, ¿cuál es el funcionamiento práctico de este esquema? En primer lugar definimos un "objeto fichero" del tipo apropiado *[i][o]fstream* para después proceder a su apertura, como por ejemplo

```
ofstream miFicheroEnModoEscritura;
miFicheroEnModoEscritura.open( "fichero.txt", ios::out );
```

O en un solo paso:

```
ofstream miFicheroEnModoEscritura( "fichero.txt", ios::out );
```

donde el primer argumento representa el nombre del fichero a abrir, mientras que el segundo indica el "modo" de apertura, de tal forma que en caso de un objeto *ofstream* podría ser *ios::out* (output mode) ó *ios::app* (append mode), mientras que en el caso de un objeto *ifstream* sería *ios::in* (input mode); un objeto de tipo *fstream* podría aplicar cualquiera de los modos[13]. Tras las operaciones con el objeto fichero éste se cerraría mediante la aplicación de la correspondiente función miembro

```
miFicheroEnModoEscritura.close();
```

En realidad los objetos *[i][o]fstream* cuentan con variedad de funciones miembros, entre las que podríamos destacar:

```
fstream miFichero;
// apertura de un fichero (los modos se pueden disyuntar)
miFichero.open( "fichero.txt", ios::in | ios::out );
// coloca el puntero en una posición determinada del fichero,
//  donde POS sería sustituido por beg (inicio: POR DEFECTO),
// cur (posición actual) o end (final fichero)
miFichero.seekg( long posicion, ios::POS );
// devuelve  la posición actual en un fichero
//  desde el inicio del mismo
miFichero.tellg();
// cierra fichero
miFichero.close();
```

Por supuesto los objetos *[i][o]fstream* tienen acceso a las funciones miembros de los *streams* detalladas en parágrafos anteriores, a efectos del uso de formateadores de entrada/salida, etc. Naturalmente, también, los

[13] En realidad la función miembro "open" consta de un tercer argumento, que podríamos calificar como de **acceso**, y que mantiene una correspondencia con los conocidos atributos de los ficheros: normal, read-only, hidden y system. Por defecto tal argumento está establecido en la posición de "normal".

operadores '<<' y '>>' han sido sobrecargados para su uso por objeto ficheros, de tal forma que el código

```
ofstream miFichero( "clientes.txt", ios::app );
miFichero << "Piratas Informáticos del Mediterráneo S.L.";
```

responde a la interpretación: inserción de un objeto "cadena" en un objeto *ofstream*. Como veíamos con los *streams*, no debemos preocuparnos de señalar expresamente el tipo del objeto a ser insertado o extraído.

De la misma forma que ocurre con los ficheros, en el fichero "**strstream.h**" se pueden encontrar distintas clases para el manejo de las "cadenas" o arrays de caracteres.

Razones de espacio y el ánimo de no oprimir en demasía al lector causan que no pueda ser más explícito en lo que ejemplos de ficheros y "strings" se refiere. Existe, por otro lado, la creencia generalizada de que los *streams* sólo sirven para el manejo elemental de las operaciones de i/o, debiendo acudir a una librería de clases especializada (incluida o no en una librería para la creación de GUI's) para encontrar desarrollos complejos de estas operaciones. Bien, ésta es parte de la verdad: en muchos de los casos demuestra únicamente que las librerías por defecto son fácilmente extensibles, mientras que las otras veces indica la gran flexibilidad de C++ para implementar desarrollos "propietarios" de interfaces "a medida".

Mi consejo: sólo se aprende a programar en C++ programando y estudiando buenos programas en C++, y puesto que la librería **iostream** es una excelente muestra de cuanto puede dar de sí el lenguaje, a la vez que explicita las facilidades de éste para la OOP, sólo puedo repetir: lean, lean, estudien, lean, lean, amplien.

7

CUANDO IRRUMPEN LOS "CONSTRUCTORES"

En este capítulo despacharé el sustrato básico que sostiene a *constructores* y *destructores*, aunque teniendo siempre en mente las dificultades de comprensión que este apartado, en un primer contacto, suele generar. Cualquier programador de un lenguaje tradicional, y por ende de C, no admite sorpresas en cuanto a lo que un determinado código puede ofrecer: no hay más ni menos que lo exactamente escrito. Para infortunio de estas personas, sin embargo, una insignificante porción de código en C++ puede encerrar, bajo su inocente apariencia, una compleja y sutil maraña de operaciones invisibles no reflejadas expresamente en el código. Realmente esto suele poner muy nerviosos a ciertos programadores, que siempre tienen la sensación de no controlar del todo lo que está ocurriendo "de verdad" cuando se ejecuta su código. Este nerviosismo se acrecienta aún más, si cabe, cuando se utiliza un depurador y el desarrollador novicio observa, anonadado, cómo en un determinado momento pasa a encontrarse, sin llamada expresa, en el cuerpo de un constructor; cómo, también, ciertas líneas nunca se ejecutan porque en el momento menos esperado aparece inmerso en el cuerpo de un destructor; cómo, en otra ocasión, se ejecuta una extraña función apuntada por un handler cuya existencia desconocía. Con tales precauciones desenfundadas intentaré, pues, explicar cómo se construyen y destruyen objetos en la práctica, de tal forma que supeditaré el rigor formal al didactismo, aun a riesgo de resultar descorazonadamente simple. Pongámonos sin más dilación el casco y pasemos al interior.

INICIALIZACIÓN DE OBJETOS

En capítulos anteriores revisamos brevemente conceptos tales como el puntero implícito *this*, las funciones y datos miembros, el calificativo *const*, etc. O sea, hemos empezado a escudriñar la estructura interna de las clases, siendo así que parece que por un lado tenemos la "filosofía de objetos" y por otro la "sintaxis de clases". Sí, sí, esto está muy bien, Pero, ¿qué pasa con la transición práctica desde la *clase* a los *objetos*? O mejor: dado que los

objetos son *instancias* de clases, ¿cómo se *instancian* éstas para producirlos? ¿Cómo, en definitiva, se "construyen" objetos? Veamos el problema con un ejemplo:

```
class Direccion {
private:
        char* calle;
        long cp;
        char* ciudad;
        // …
};

class Persona {
private:
        char* nombre;
        Direccion* direccion;
        // …
};
```

Tenemos que la clase Persona contiene un puntero a un objeto de la clase Direccion. Examinemos el funcionamiento de una variable de tipo predefinido: si en un bloque local codificamos

```
float objetoDeTipoPredefinido;
```

el compilador automáticamente inicializará el objeto, reservando espacio para su alocación en memoria, a la vez que procederá a su destrucción al salir del ámbito local en que ha sido declarado. ¿Qué ocurre, sin embargo, con los objetos de tipo definido-por-el-usuario mediante clases? ¿cómo se inicializan estos? La respuesta es clara: inicializando la totalidad de sus miembros. Y esto, ¿cómo se lleva a cabo? Bueno, una solución inmediata podría ser dotar de funciones de inicialización para cada miembro que serían llamadas secuencialmente, como por ejemplo:

```
class Direccion {
public:
        void estableceCalle( const char* miCalle ) {
                calle = new char[ strlen( miCalle) + 1 ];
                strcpy( calle, miCalle );
        }
        void estableceCp( const long miCp) {
                cp = miCp;
        }
        void estableceCiudad( const char* miCiudad ) {
                ciudad = new char[ strlen( miCiudad ) + 1 ];
                strcpy( ciudad, miCiudad );
        }
        // …
};

class Persona {
public:
        void estableceNombre( const char* miNombre ) {
```

```
                nombre = new char[ strlen( miNombre) + 1 ];
                strcpy( nombre, miNombre );
        }
        void estableceDireccion( Direccion* miDireccion ) {
                direccion = miDireccion;
        }
        // …
};
```

de tal forma que la inicialización de un objeto de tipo *Persona* debería ser realizada mediante, por ejemplo, la siguiente secuencia compleja:

```
Direccion miDireccion;
miDireccion.estableceCalle( "Orense, 36" );
miDireccion.estableceCp( 28020 );
miDireccion.estableceCiudad( "Madrid" );
Persona miPersona;
miPersona.estableceNombre( "John Doe" );
miPersona.estableceDireccion( &miDireccion );
```

Pero ¡no, esto es ridículo!. Imaginemos una secuencia compleja de objetos contenidos en otros objetos que a su vez contienen a otros: para inicializar un simple objeto deberíamos conocer cómo se inicializan multitud de otros objetos, y esto atenta contra el núcleo de la OOP: cada objeto debe encapsular sus propios métodos de inicialización. O sea, ¿quién ha de saber más de inicializar un objeto que el objeto en sí? Una solución más depurada podría ser la inclusión en cada objeto de una función especial de inicialización, que podríamos denominar *init*:

```
class Direccion {
public:
        void init(  const char* miCalle,
                        const long miCp,
                        const char* miCiudad)
        {
                calle = new char[ strlen( miCalle) + 1 ];
                strcpy( calle, miCalle );
                cp = miCp;
                ciudad = new char[ strlen( miCiudad ) + 1 ];
                strcpy( ciudad, miCiudad );
        }
        // …
};

class Persona {
        void init(  const char* miNombre = "",
                        const char* miCalle = "",
                        const long miCp = 0,
                        const char* miCiudad = "" )
        {
                nombre = new char[ strlen( miNombre ) + 1 ];
                strcpy( nombre, miNombre );
                Direccion miDireccion;
```

```
                    miDireccion.init( miCalle, miCp, miCiudad );
                    direccion = &miDireccion;
            }
            // …
};
```

De esta forma la inicialización de un objeto de tipo *Persona* se codificaría de la siguiente guisa:

```
void evaluaBushGate()
{
        Persona candidatoDemocrata;
        // en la siguiente línea el resto de argumentos
        //  se aplica por defecto
        candidatoDemocrata.init( "Bill Clinton" );
        // realiza algún tipo de proceso
}
```

Lo cierto es que las cosas se han simplificado bastante. Pero en este esquema hay algo que no funciona: ¿Qué ocurre con el objeto candidatoDemocrata cuando la función en cuyo ámbito local se ha declarado termina? ¿Debemos implementar una nueva función, quizás denominada *fin*, que deba ser expresamente llamada antes del fin del ámbito en que se declara un objeto local? ¿Qué tipo de desastre puede ocurrir si usamos el objeto definido antes de inicializarlo? ¿No estamos confundiendo, por otra parte, la inicialización de un objeto con la asignación de valores a sus datos miembros? Veamos qué ocurre con los objetos de tipo predefinido:

```
void hazNoSeQue()
{
        char[ 16 ] nombrePresidente;            // inicialización
        nombrePresidente = "Felipe González"   // asignación
        // …
}       // fin del ámbito local: desinicialización
        //  o destrucción de las variables locales
```

Realmente no tenemos que preocuparnos por inicializar expresamente las variables de tipo predefinido, como tampoco tenemos que expresamente desinicializarlas al final del ámbito en que se han definido. Existe, pues, una clara diferencia con respecto a nuestro anterior intento de construcción de objetos. O sea, de alguna manera el compilador se ocupa de llamar a un "constructor" para la inicialización de objetos (variables) de tipo predefinido, usando, en su caso, una suerte de "destructor" para liberar el espacio ocupado por éste al salir de un determinado ámbito. En la realidad no ocurre exactamente así, pero esto nos da una idea aproximada de un enfoque que podemos aplicar a los objetos instanciaciones de nuestras clases.

Vemos, pues, que en el caso de variables globales o estáticas (siempre de tipo predefinido) el "constructor" predefinido del compilador las inicializa a cero, mientras que las variables locales se "construyen" sin valor fijo. ¿Qué ocurre, empero, con "nuestros" objetos? ¡Exactamente lo mismo! El compilador, de hecho, provee automáticamente un constructor por defecto para cada clase. Pero, ¿qué demonios realiza este constructor y cuándo entra en acción? Bien: el constructor de un determinado objeto es llamado cuando tal objeto es definido, igual que ocurre con los objetos de tipo predefinido, y su funcionamiento es el siguiente: secuencial y recursivamente se inicializan o "construyen" los distintos datos miembros de los objetos. O sea, si tenemos

```
class Evento {
private:
        long claveEstadistica;
        Racional probabilidad;
        // …
};

class Racional {
private:
        int numerador, denominador;
        // …
};
```

la siguiente codificación

```
Evento tiradaDeDado;
```

causará que se aplique el siguiente esquema de "constructores" por defecto: primero se "construirá" por el compilador, de la forma usual, un objeto de tipo *long*, para seguidamente construir un objeto de tipo *Racional*, para lo cual, recursiva e iterativamente, se aplicarán los constructores por defecto para cada uno de los datos miembros de esta última clase: o sea, se aplicarán dos "constructores" predefinidos para los dos objetos de tipo *int* correspondientes a los identificadores numerador y denominador. Naturalmente, y de forma parecida a como operan los constructores, el compilador proporciona también unos "destructores" por defecto que oportunamente liberarán los objetos construidos. Vemos, de esta manera, que el tratamiento de construcción de objetos de cualquier tipo es homogéneo. De todas formas este planteamiento de constructores implícitos no nos soluciona de forma clara el problema, pues es posible que nosotros deseemos inicializar un objeto de forma distinta a lo establecido *por defecto*. Y la verdad es que si no pudiéramos construir un objeto a nuestro gusto -dentro de ciertos límites razonables-, la flexibilidad de que C++ tanto hace gala en el diseño de nuevos tipos de datos abstractos quedaría peligrosamente emborronada. Bien, lo cierto es que sí podemos: y es aquí donde con propiedad entran en liza los **constructores**, que, en definitiva, son funciones miembros de una determinada clase, con una particular sintaxis, que proporcionan los métodos para construir o instanciar objetos de tal clase. Los constructores poseen como identificador el mismo de la clase a que pertenecen, y no poseen valor ni tipo alguno de retorno. Veamoslo con un ejemplo:

```
class ControlMilitar {
```

```
private:
        char sexo;
        int edad;
public:
        ControlMilitar();       // no devuelve valor alguno,
                                //  ni siquiera 'void'
};

ControlMilitar::ControlMilitar()
{
        sexo = 'V';
        edad= 18;
}
```

de tal forma que la línea de código siguiente,

```
ControlMilitar ejercicioActual;
```

donde se define un objeto de tipo ControlMilitar, origina que se produzca una llamada al *constructor* implementado por nosotros, de manera que los datos miembros del objeto ejercicioActual se inicializan a 'V' y '18'. Hemos sustituido, de hecho, al constructor "implícito" del compilador para esta clase. Pero la situación sigue sin ser totalmente satisfactoria: ¿y si deseáramos parametrizar la inicialización de nuestro objeto? ¿por qué limitarnos a un rango de valores por defecto? Bien, es razonable suponer que, dado que los constructores son en realidad funciones miembros, admitirán argumentos en tipo y número arbitrarios. Recodifiquemos, a efectos de ejemplo práctico, la clase anterior:

```
class ControlMilitar {
private:
        char sexo;
        int edad;
public:
        ControlMilitar( char miSexo, int miEdad )
        {
                sexo = miSexo;
                edad = miEdad;
        }
};
```

En esta ocasión un objeto de nuestra clase únicamente podría construirse con la siguiente notación:

```
ControlMilitar ejercicioActual( 'V', 17 );
```

otorgándonos más control sobre el estado interno del objeto. ¿Qué ocurre, sin embargo, si con la anterior definición de la clase codificamos la siguiente inofensiva línea?

```
ControlMilitar ejercicioActual;
```

El lector podría presumir que, dado que hemos proporcionado un constructor con un número y tipo de argumentos que no encaja con esta definición del objeto, entrará en acción el *constructor implícito* del

compilador comentado líneas atrás. El lector, en efecto, podría sostener tal presunción, pero se equivocaría: el hecho que hayamos definido uno o más constructores (pues, como el lector ya habrá inferido, merced a la sobrecarga de funciones y dado que el constructor es básicamente una función, se pueden definir distintos constructores para una misma clase) anula tal constructor implícito, por lo que el uso del objeto tras la línea anterior podría conducir a un resultado impredecible, pues éste no ha sido propiamente inicializado. ¿Quiere esto decir que debemos expresamente implementar en nuestra clase un constructor sin argumentos? Bien, en principio podríamos pensar que nadie nos obliga por la fuerza a conducirnos así: sólo habría que tener la precaución de codificar siempre la inicialización de los objetos con el número y tipo de argumentos significados en nuestros constructores. Veamos que, de nuevo, sin embargo, esto no funciona: imaginemos que deseamos inicializar un array de objetos del tipo

```
ControlMilitar arrayDeControlesMilitares[ 10 ];
```

¿Qué constructor usará el compilador para inicializar los objetos del array? ¡El constructor sin argumentos! Pero no hemos definido tal constructor y el implícito por defecto ha sido "cancelado" al implementar otro constructor, por lo que una posible, tediosa y a veces posiblemente impracticable solución sería explicitar la inicialización de cada objeto con el constructor disponible, como por ejemplo:

```
ControlMilitar arrayCM[ 2 ] = {     ControlMilitar( 'V', 17 ),
                                    ControlMilitar( 'V', 18 ) };
```

En realidad estamos "parcheando" una evidente carencia, cuando lo más fácil es suplirla: dotemos, pues, a la clase con tal constructor por defecto:

```
class ControlMilitar {
public:
        ControlMilitar() {};
        // ...
};
```

¿Y ya está? -preguntará el asombrado lector-, ¿esto es todo? ¿una función sin argumentos y con un cuerpo vacío, que no hace absolutamente nada? ¿Dónde queda aquí la inicialización de los datos miembros de nuestra clase? Tranquilidad, amable lector. Resulta que el lenguaje C++ establece que si un constructor no inicializa de forma expresa a los miembros de su clase (tanto como a sus clases base, si éstas existen, como veremos más adelante), automáticamente entrará en acción el constructor por defecto para cada uno de ellos. En el caso que nos ocupa, como quiera que el constructor ControlMilitar() no inicializa expresamente ningún miembro, el resultado es que al ejecutarse produce que primero se aplique el "constructor" de tipo *char* a la variable *sexo*, y después el "constructor" de tipo *int* a la variable *edad*, que quedarán, como variables locales, en un estado indefinido. O sea, el mismo comportamiento que si no hubiéramos definido ningún constructor en la clase y se aplicara el constructor implícito *por defecto*. De hecho la anterior codificación viene a decir: "queremos uno o más constructores concretos para nuestros objetos, pero sin renunciar a la aplicación del constructor por defecto", y ello conseguido con un mínimo esfuerzo escritor.

Bueno: parece que el panorama se va aclarando. Examinemos, de cualquier forma, una postrera posibilidad: un constructor con argumentos por defecto. Veámoslo:

```
class ControlMilitar {
public:
        ControlMilitar( char = 'V', int = 18 );
private:
        char sexo;
        int edad;
        // ...
};

ControlMilitar::ControlMilitar( char miSexo, int miEdad )
{
        sexo = miSexo;
        edad = miEdad;
}
```

Quedarían así grandemente resueltas nuestras preocupaciones. Pero miremos con más detenimiento en el cuerpo de nuestro constructor. ¿Qué vemos? Pues la simple asignación de dos objetos de tipo *char* e *int*. Nada de construcción de objetos. Recordemos la regla expuesta un poco más atrás: si en un constructor un miembro no se inicializa de forma expresa, cual es éste el caso, entrará en acción el constructor por defecto para tal miembro. O sea, que la aplicación de este último constructor originará que, antes de ejecutarse el cuerpo del mismo, se construyan los objetos de tipo *int* y *char*, para después proceder a la asignación a ambos de determinados valores. Estamos, de hecho, duplicando innecesariamente el trabajo, pues lo ideal sería que el constructor asignara directamente los valores deseados. En realidad existe una sintaxis específica a este fin:

```
ControlMilitar::ControlMilitar( char miSexo, int miEdad )
        : sexo( miSexo ), edad( miEdad) { }
```

Los dos puntos tras el constructor indican que seguirá, antes de que comience el cuerpo del mismo, una lista de inicializadores de los datos miembros de la clase (y también de las clases base, en su caso). Vemos, también, que éste constructor posee un cuerpo vacío. ¡Naturalmente! Este código indica expresamente cómo deben ser construidos los objetos miembros, a la vez que, en la misma operación, les asigna los pertinentes valores, aún antes de comenzar la ejecución del cuerpo de la función constructora que, en este caso, no aportaría nada nuevo. En lo posible debemos usar la sintaxis de inicialización de miembros en lugar de la de asignación en constructores, aunque en algunas ocasiones esto será obligatorio: los datos miembros *referencias* o *const* no se pueden asignar, sino sólo inicializar.

Repasemos esta curiosa sintaxis de inicialización en constructores. ¿Se ha fijado el lector si los datos miembros aparecen en la lista de inicialización en el mismo orden en que aparecen declarados en el protocolo de descripción de su clase? ¡Demonios! ¿Qué es esto? -inquirirá con cierta razón el lector- ¿Una nueva sutileza? Bien, algo parecido: ¿qué orden, por ejemplo, sigue un constructor por defecto en la "construcción" de los datos miembros de su clase? Con cierta lógica sigue el orden de aparición de la declaración de los miembros en la clase dada. ¿Qué ocurre, sin embargo, si cambiamos este orden en nuestra lista de inicialización? ¿Podemos así elegir el orden de construcción de nuestros datos miembros? Bien, la verdad es que no. El orden de "construcción" de los miembros permanecerá, en cualquier caso, invariante. ¡Vaya! Y, ¿por qué? Pues, básicamente, porque los *destructores* (algo así como el reverso de los constructores, y que veremos enseguida) siguen un orden opuesto al de los constructores asociados, de tal forma que el sistema debe controlar este último orden, de forma que si tal dependiera de nuestro capricho el sistema sufriría una inútil penalización. ¿Tiene algún sentido, entonces, situar los inicializadores de la lista en el orden correcto? ¡Por supuesto! Es una forma de evitar erróneas interpretaciones en la evaluación del orden de los constructores, reforzando, de paso, la idea que el programador sabe exactamente qué es lo que está haciendo.

Revisemos ahora otras formas usuales de construcción de objetos de tipo predefinido:

```
char tuLetra = 'Ñ';          // constructor
char miLetra = tuLetra;                  // constructor
miLetra = tuLetra;           // ¿constructor?
```

Las dos primeras líneas construyen dos objetos de tipo *char* y, en el mismo acto, copian el valor situado a la derecha del signo '=' a cada uno de ellos. En la primera línea, en puridad, se construye primero un objeto temporal de tipo *char* a partir del carácter 'Ñ', y seguidamente se copia tal objeto en el objeto de nueva construcción con identificador *tuLetra*. En la segunda línea se produce, sin más, una construcción con copia simultánea. A este tipo de constructores se les denomina, víctimas de una imaginación léxica sorprendente, *constructores de copia*. Examinemos ahora la última línea de código: se sustituye la representación interna del objeto a la izquierda del operador '=' con la del objeto a la derecha de éste. En definitiva, un objeto se *asigna* a otro pero, como el atento lector ya habrá notado, no se construye ningún objeto, sino que se trata de una operación entre objetos "ya construidos"; una operación correspondiente al *operador de asignación (operator=)*. De hecho, para distinguir entre *asignación* e *inicialización* debemos preguntarnos: ¿se ha construido algún objeto? En caso negativo podríamos inferir que se trata de una *asignación*.

Al igual que hemos visto anteriormente, si no proporcionamos de forma expresa constructor de copia y operador de asignación a una determinada clase, el compilador suplirá tales funciones implícitamente con el siguiente comportamiento: en el caso del constructor de copia, se construirá un objeto

y seguidamente se inicializarán sus datos miembros con los mismos valores del objeto a copiar; en el caso del operador de asignación simplemente se copiarán los valores de los datos miembros de un objeto a otro. Bueno, esto parece suficiente -podría afirmar, ya cansado, el lector- ¿Existe alguna razón para codificar nuestras propias versiones de tales funciones? Sí, a veces. Échenle un ojo, si no, al siguiente ejemplo:

```
class Persona {
public:
        Persona( char* miNombre = 0, int miEdad = 0 )
                : edad( miEdad )
        {
                if ( miNombre ) {
                        nombre = new char[ strlen( miNombre ) + 1 ];
                        strcpy( nombre, miNombre );
                } else {
                        nombre = new char[ 1];
                        *nombre = '\0';
                }
        }
private:
        int edad;
        char* nombre;
        // …
};

Persona   primeraPersona( "Luis", 31 ),
                terceraPersona( "Antonio", 27 );
// constructor de copia
Persona   segundaPersona = primeraPersona;
// operador de asignación
terceraPersona = primeraPersona;
```

De acuerdo con lo dicho, los datos miembros de los objetos segundaPersona y terceraPersona son los mismos que los del objeto primeraPersona. Esto quiere decir, por ejemplo, que el miembro edad tiene el valor de '31' en todos los objetos. ¿Qué ocurre, empero, con el miembro nombre? Pues que en los objetos se ha copiado el valor del puntero que apunta a la cadena de caracteres "Luis" del objeto primeraPersona. O sea, los tres miembros de los objetos apuntan a la misma cadena. Pero es muy posible que esto no sea lo que deseábamos conseguir. Imaginemos que si, por ejemplo, destruimos uno de los objetos y se "destruye" la cadena, los restantes objetos quedarán en una situación cuando menos curiosa, si no desastrosa. ¿Cómo podríamos solucionar esta dificultad? Pues codificando nuestras propias funciones:

```
class Persona {
public:
        Persona( const Persona& miPersona      )
        {
                edad = miPersona.edad;
                nombre = new char[ strlen( miPersona.nombre ) + 1 ];
                strcpy( nombre, miPersona.nombre );
        }
        Pesona& operator=( const Persona& miPersona )
        {
                if ( this == &miPersona )//chequea autoasignación
```

```
                return *this;
        edad = miPersona.edad;
        delete [] nombre;    // desaloja espacio almacenamiento
                                // libre antigua cadena
        nombre = new char[ strlen( miPersona.nombre ) + 1 ];
        strcpy( nombre, miPersona.nombre );
        return *this;
    }
    // sigue resto de la descripción de la clase
};
```

Vemos también, de paso, un primer ejemplo sencillo de sobrecarga de operadores en el ámbito de una clase, que ampliaremos en un capítulo posterior. Bien, en definitiva, con esta codificación conseguimos que se copien las cadenas de un objeto a otro, y no sólo los punteros.

Consideremos, para terminar este parágrafo, la construcción de un objeto apuntado por un puntero:

```
Persona* punteroAPersona;
punteroAPersona = new Persona( "Roque", 52 );
```

DESTRUCTORES

Así como existen funciones constructoras para nuestros objetos, de la misma forma existen sus opuestas: las encargadas de "desinicializar" lo inicializado: los *destructores*. La única diferencia conceptual es que mientras que puden ser definidos varios constructores, sólo es posible definir un destructor por clase, lo cual es lógico, pues existen, por ejemplo, infinidad de figuras a ser construidas mediante la papiroflexia, pero sólo una forma de quemarlas.

Si no dotamos de destructor expreso a una clase, el sistema, al igual que ocurría con los constructores, asumirá uno por defecto, y cuyo funcionamiento consistirá exactamente en las operaciones inversas de las que realizaría el constructor por defecto. Sin embargo este planteamiento no serviría, verbigracia, para nuestro último ejemplo, pues en los constructores alojamos una cadena en el espacio de memoria de almacenamiento libre, mediante el operador *new*, y si simplemente dejamos que actúe el destructor por defecto, únicamente se "destruirá" el puntero, pero no será liberada la memoria utilizada para la cadena. ¿Qué debemos, pues, hacer? Codificar nuestro propio destructor, naturalmente. Los destructores no poseen tipo de retorno ni argumentos y ostentan como identificador el mismo que el de la clase antecedido por la tilde ~. Veámoslo:

```
class Persona {
public:
    ~Persona()
    {
        delete [] nombre;
```

```
        }
        // sigue resto descripción de la clase
};
```

Al igual, también, que ocurría con los constructores, si en un destructor no se desinicializa expresamente un dato miembro, automáticamente actuará el destructor por defecto para el mismo.

Pero, entonces, ¿cuándo debe ser usada una función "destructora"? Bien, lo cierto es que raras veces es necesario llamar expresamente a un destructor. La inmensa mayoría de las veces éste es usado de forma implícita por el compilador, produciéndose una llamada automática al mismo en cualquier momento desde la última aparición de un objeto y el fin del ámbito del mismo.

UN SENCILLO CONTROL

Este es un buen momento para explicitar una pequeña triquiñuela, muy habitual, por otra parte, en los textos prácticos de introducción al lenguaje: en tanto el lector no conozca con precisión la mecánica del lenguaje con respecto a estas particulares funciones miembros, es francamente recomendable incluir líneas del tipo *imprime "Construcción del objeto X mediante el constructor de copia"*, o *imprime "Destrucción del objeto Y"*. Al ejecutarse el código el lector podrá perfectamente apreciar en qué momento se activan los constructores y destructores.

8

DE LA OBJETIVIDAD DE LO RACIONAL

En fin, hasta ahora hemos visto bastante de teoría y muy poco de código práctico. Y es hora de que pongamos las manos en la masa. Por eso éste preámbulo será excepcionalmente corto. Tan corto que ya se acabó.

DEFINAMOS UN NUEVO TIPO: LA CLASE "RACIONAL"

En los capítulos precedentes han aparecido frecuentes referencias, acompañadas por cortos fragmentos de código, a una clase denominada *Racional* que, en síntesis, pretendía encapsular las características del conjunto matemático de los números Racionales en una clase, constituyendo lo que se denomina un ADT (*Abstract Data Type*: Tipo de Dato Abstracto). Bien, ya hemos llegado a un punto del relato en que podemos abordar la construcción de dicha clase, e intentaremos hacerlo con cierto método, a la vez que explicando cada decisión de diseño. Vayamos a ello.

¿Qué es, en definitiva, un número racional? Bien, la definición matemática comprehensiva sería la siguiente (y obviemos la notación específica):

Racionales = { **x**/**y** | **x** es un número Entero e
y es un número Entero }

O sea, un racional es un número representado por dos números enteros (ojo: enteros matemáticos) separados por el operador de división: lo que los niños conocen como "quebrado". El número a la izquierda del operador '/' se denomina *numerador*, y el otro *denominador*. En esencia, pues, se trata de un par de números relacionados por un operador. Bien, ya podemos intentar un primer acercamiento a la compósición interna de los objetos de tipo *Racional*:

```
class Racional {
private:
        int numerador;
        int denominador;
```

```
};
```

Hemos declarado los datos internos como *privados*, siguiendo los esquemas de encapsulación y ocultación de la información preconizados por la OOP. En general los datos miembros de una clase se declararán *private*, dejando el interfaz de cliente (la parte *public*) únicamente para funciones miembros. El cliente de la clase[14] no tendrá así dudas, entre otras cosas, sobre añadir o no paréntesis funcionales a los miembros del interfaz público (situación resuelta, por otro lado, en el lenguaje Eiffel por la no diferenciación sintáctica entre datos y funciones miembros). Pensemos también que podríamos cambiar, por ejemplo, el tipo de estos datos (pasándolos a *long*), o añadir un nuevo dato miembro: su acceso exclusivo a través de funciones miembros públicas nos asegura que en caso de tales cambios (que serán oportunamente reflejados en la implementación de tales funciones) el código de los clientes de la clase no deberá ser en absoluto variado.

¿Qué ocurre, empero, con el operador de división? ¿Realmente no forma parte de la representación interna de los objetos *Racionales*? ¡No, diantre! ¡Únicamente forma parte de su representación *visual* o *gráfica*! ¿Y no es, por otro lado, el verdadero número racional el cociente resultante de dividir numerador por denominador? Ya hemos visto que no, al definir el conjunto de los *Racionales*. En realidad tal número decimal podría considerarse como la conversión a *float* de un *Racional*. Como vemos, es conveniente aclarar las ideas ante la siempre difícil tarea de definir una clase, descubriendo a veces que lo obvio no es lo más apropiado.

De acuerdo, pensará el lector: aceptemos la privacidad de los datos internos. Inmediatamente, entonces, deberemos dotar a la clase de funciones públicas de acceso a tales datos, para permitir su inicialización y modificación. ¡Tranquilo, amable lector! ¡Cada cosa en su momento! Lo primero que debemos hacer, como ya creemos tener completa la representación interna de la clase, es dotar a ésta de constructores y destructor apropiados. Veamos primero los constructores.

Como vimos en el capítulo anterior, si no dotamos de un constructor expreso a nuestra clase, el sistema proveerá un constructor implícito por defecto (sin argumentos), lo que en este caso no parece suficiente. En general podemos pensar que la construcción de un objeto *Racional* requerirá dos argumentos, justos los correspondientes a los datos miembros *numerador* y *denominador*:

[14] Hay que tener en cuenta que C++ enfatiza sobremanera la reutilización del código, fundamentalmente mediante la derivación de clases. Debe pensarse, en esta tesitura, que el código de nuestras clases, al menos en lo que se refiere al interfaz o archivos de cabecera, será revisado por desarrolladores o usuarios, a quienes genéricamente denominaremos *clientes* y cuya vida nos guardaremos mucho de complicar innecesariamente. Son clientes, también, los objetos que hacen uso del interfaz de nuestra clase.

```
class Racional {
public:
        Racional( int, int );
private:
        int numerador;
        int denominador;
};
```

Como el lector notará, estamos diseñando con cruel lentitud la clase ejemplo, pero existen tres razones básicas para ello: primera: esto es una introducción gentil al lenguaje; segunda: el diseño eficiente de clases, como ya ha quedado dicho, es ciertamente difícil; tercera: lo obvio en C no tiene por qué serlo en C++ (y, de hecho, en lo que se relaciona con las clases, la mayoría de las veces no lo es).

Bien, ya hemos declarado un constructor. ¿Implementamos ahora su definición? ¡No! Vamos a despachar primero un intento de interfaz completo de la clase (sólo declaraciones de datos y funciones miembros, *incluso de las inline*), pues esto allanará grandemente su implementación posterior. Intentaremos, pues, seguir esta norma, aunque sin fanatismos, pues en C++ muchas veces hay que volver la vista atrás para desandar lo andado[15], así como adelantar para adivinar lo pasado. Sigamos, entonces, con los constructores. Con arreglo a lo codificado podemos construir objetos *Racional*es de la siguiente forma:

```
Racional dosQuintos( 2, 5 );
Racional ochoTercios( 8, 3 );
Racional nueve( 9, 1 );         // ¡Todos los enteros son Racionales!
Racional cero( 0, 1 );
```

aunque, ¿qué ocurre si codificamos lo siguiente?:

```
Racional fraccion;   // ¡error!:
                                        //  no existe el contructor Racional()
```

Pues que, al haber anulado el constructor implícito con nuestro constructor de dos argumentos, el compilador flagelará como error tal línea. La solución más inmediata sería la declaración expresa de un constructor por defecto:

```
class Racional {
public:
        Racional();
```

[15] En el desarrollo de sistemas software en C++ es frecuente la aplicación de la máxima "**analizar un poco, diseñar un poco, codificar un poco**", de tal forma que el proceso de creación se convierte en un iterativo refinamiento de cada etapa en sucesivos ciclos, involucionándose éstos de manera que se quiebra el tradicional esquema de "cascada" de fases.

```
        Racional( int, int );
private:
        int numerador;
        int denominador;
};
```

Tal constructor por defecto posibilitaría dos codificaciones. La primera, con el cuerpo vacío, de la siguiente guisa:

```
Racional::Racional() { ; }
```

inicializará los datos miembros de la forma usual en que el compilador inicializa variables locales de tipo *int*: dejándolas con un valor indefinido. Quizá esto no sea lo más apropiado. La segunda codificación proporcionaría valores por defecto a nuestros datos miembros:

```
Racional::Racional() : numerador( 0 ), denominador( 1 ) {}
```

Vemos que la construcción de un racional sin argumentos equivaldría, así, a la construcción del racional '0/1'. Se ha preferido, por otro lado, la lista de inicialización frente a la asignación de los datos miembros en el cuerpo del constructor, según se explicó en el anterior capítulo, siendo ésta una preferencia que el lector deberá tomar como norma.

Tenemos, pues, dos constructores: uno con dos argumentos y otro por defecto. Pero examinemos el constructor por defecto: en realidad inicializa los datos miembros de la misma forma que lo haría el constructor con argumentos, lo único que con unos valores predefinidos. ¿Por qué no unir estos dos constructores en uno solo con dos argumentos a los que se asignarían parámetros por defecto? Nuestra clase quedaría ahora así:

```
class Racional {
public:
        Racional( int = 0, int = 1 );
        ~Racional() {};
private:
        int numerador, denominador;
};
```

Hemos incorporado, también, el destructor con el cuerpo vacío, pues no hay nada concreto que hacer para desinicializar los objetos sino dejar que el "destructor" de los datos miembros de tipo *int* actúe como lo haría con variables locales "normales". Ahora podríamos codificar lo siguiente:

```
Racional unQuinto( 1, 5 );
Racional miFraccion;            // equivale a miFraccion( 0, 1 )
Racional nueve( 9 );            // equivale a nueve( 9, 1 )
```

Por supuesto que otra posible solución hubiera sido declarar parámetros por defecto en el constructor con dos argumentos y mantener, a la vez, el

constructor sin argumentos. Pero, claro, esto no es una solución: es un grave error. Así las cosas la siguiente línea:

```
Racional miObjetoRacional;          // error: ambigüedad
```

ocasionaría un error por ambigüedad en la llamada al constructor: ¿se llama al constructor de dos argumentos con los dos parámetros asumidos por defecto? ¿o bien se llama al constructor sin argumentos? Dejemos, pues, en este caso, un solo constructor.

El lector debe ser, no obstante, cuidadoso a este respecto, pues si bien en ARM se establece que un constructor con todos sus argumentos establecidos por defecto es un constructor por defecto, algunos compiladores actuales se niegan a aceptar este hecho, por lo que, en tales casos, habría que codificar un constructor expreso sin argumentos que sería utilizado, por ejemplo, en la construcción de un array de objetos. No debemos permitir, de cualquier manera, que las deficiencias de algunos compiladores enturbien nuestros esquemas de codificación[16].

¿Qué hubiera ocurrido, por otro lado, si el constructor hubiera sido declarado en la sección *private* de la clase? Pues que los objetos no podrían acceder a él y, por tanto, no podrían ser construidos, con lo que las anteriores líneas habrían procurado sendos errores en compilación. ¿Quiere decir esto que los constructores siempre habrán de ser *públicos*? ¡No, ni mucho menos! ¡Líbreme Pessoa de tales afirmaciones categóricas en C++! Los constructores *privados* se dan, por ejemplo, en clases anidadas, en estructuras *carta-sobre* o formas de *Singleton*, como ya veremos -repitan ustedes- más adelante.

"Otra cosa", continuará el lector: ¿Tenemos cada vez que reescribir la descripción de la clase para añadir miembros? ¿No hay forma de añadírselos, como apéndices, tras la definición de la clase? ¡No, ciertamente!

[16] Quizá lo más práctico, en este y otros casos de desajuste entre ARM y nuestra implementación concreta de C++, sea encerrar el código desafortunadamente generado entre condicionales del preprocesador, como por ejemplo:

```
class Racional {
public:
#ifdef CPP_NOCOMP
       Racional() {}
#endif /* CPP_NOCOMP */
// sigue resto clase
};
```

pudiendo así aislar, en lo posible, tales deficiencias. En lo que a parámetros por defecto en constructores se refiere, Borland C++ 3.X y 4.0 no plantean ningún problema.

Pensemos que si una clase se pudiera modificar con posterioridad a su definición todo nuestro esquema de acceso se vendría abajo, pues bastaría para ello que el cliente "añadiera" una función con acceso al miembro de la clase que deseara, sin encontrar restricción alguna. Sigamos.

Bien, ya sabemos que el sistema proporciona a nuestra clase un constructor de copia y un operador de asignación implícitos. La pregunta es: ¿son suficientes para nuestros propósitos, en lo que respecta a la clase *Racional*, o más bien deberíamos definir nuestras propias versiones? Dado que nuestra clase no maneja memoria del almacenamiento libre, parece que lo que proporciona el sistema es suficiente[17]. De esta manera podemos escribir:

```
Racional dosTercios( 2, 3 ), tuFraccion( 7 );
Racional miFraccion = dosTercios;       // constructor de copia:
                                                    //  miFraccion "vale" 2/3
tuFraccion = miFraccion;                // operador de asignación:
                                                    //  tuFraccion "vale" 2/3
```

SOBRECARGA DE OPERADORES

De acuerdo: ya tenemos la representación interna y distintas formas de construir los objetos. ¿Qué sigue ahora? Bien, lo siguiente es completar el interfaz de cliente de la clase. O sea, dotar a la clase de métodos públicos que permitan la aplicación de sus instanciaciones (objetos) de una forma similar a como se haría con objetos de tipo predefinido. Esta es, en general, una labor delicada, pues exige prudencia, experiencia y una cierta comprensión de las sutilezas del lenguaje. Lo más fácil es que el novicio "hinche" el interfaz con un número elevado de funciones miembros, previendo -y aun duplicando- todos los posibles casos de uso que se le ocurran. Esto suele ser un error, pues normalmente confunde al cliente de la clase, que se siente como si hubiera entrado en un parque de atracciones que constara únicamente de montañas rusas, eso sí, diferenciadas por colores y por algún oscuro matiz que se le escapa al observador. El interfaz que nos ocupa debe ser, pues, breve y completo (algo así como una extensión informática de la célebre máxima de Gracián). Resolvamos una primera lista descriptiva de métodos necesarios:

- comparación: igualdad, mayor, menor.
- suma, resta, multiplicación y división entre objetos.
- operaciones de entrada y salida de objetos.

[17] De hecho, lo contrario puede afirmarse como regla: "siempre que una clase haga uso de la memoria de almacenamiento libre (normalmente mediante el operador *new*), el desarrollador deberá proveerla de sus propias versiones del constructor de copia y de la sobrecarga del operador de asignación".

Declaremos ahora las pertinentes funciones miembros correspondientes a tales métodos. Algo como lo siguiente:

```
typedef int boolean;

class Racional {
public:
        boolean esIgualA( const Racional& ) const;
        boolean esMayorQue( const Racional& ) const;
        Racional suma( const Racional& );
        // etc., etc.
};
```

Pero no, esto no funciona. Pensemos que para, por ejemplo, comparar dos objetos de tipo *Racional* deberíamos codificar, con arreglo a lo expuesto, lo siguiente:

```
Racional miFraccion, tuFraccion;
// …
if ( miFraccion.esIgualA( tuFraccion ))
        hazAlgunaCosa();
```

Y es que esta sintaxis no se parece a la que usaríamos, por ejemplo, con *floats*, que sería algo así como:

```
float miFloat, tuFloat;
// …
if ( miFloat == tuFloat )
        hazAlgunaCosa();
```

¿Cuál es la solución? ¡El uso de la característica de sobrecarga de operadores, naturalmente! Probemos, para empezar, con el ya preludiado operador de igualdad:

```
class Racional {
public:
        boolean operator ==( const Racional& );
        // sigue resto descripción clase
};
```

Examinemos con más detenimiento la sobrecarga declarada: se trata de una función miembro (con la especial sintaxis de los operadores) que devuelve un valor booleano y toma como argumento una referencia a un objeto constante de tipo *Racional*. ¿Un argumento? ¡Pero el operador ha de comparar dos objetos! ¿Qué pasa aquí? Bien, el argumento de nuestra función se corresponde con el objeto a la derecha del operador, mientras que el objeto correspondiente a la instanciación de la clase, que recibe el mensaje del operador (el que llama a la función operador==), se corresponde con el objeto a la izquierda del operador.

De acuerdo con lo expuesto, ahora ya podemos escribir:

```
Racional miFraccion, tuFraccion;
// ...
if ( miFraccion == tuFraccion )
        hazAlgunaCosa();
```

pero siempre teniendo en cuenta que el uso infijo del operador equivale a la siguiente notación funcional:

```
if ( miFraccion.operator==( tuFraccion ) )
        hazAlgunaCosa();
```

Probemos ahora con una operación algebráica, como por ejemplo la suma de *Racionales*. Veamos la declaración de la sobrecarga:

```
class Racional {
public:
        Racional operator+( const Racional& );
        // sigue resto descripción clase
};
```

Vemos que, de nuevo, el argumento es una referencia a un objeto constante de tipo *Racional*. Pero, ¿por qué una referencia y no un puntero o un simple objeto pasado por valor? Básicamente por una cuestión de eficiencia: si se pasara un objeto como argumento tendría lugar la siguiente secuencia: en la llamada a la función se copiaría el objeto al identificador del argumento mediante el constructor de copia (en nuestro caso mediante el constructor de copia implícito), y en algún momento desde la última línea de uso de este objeto local hasta el fin del ámbito de la función se llamaría al destructor de este objeto "copiado". Si el argumento es un puntero o una referencia no se producen estas llamadas a constructor y destructor[18], por lo que el resultado es más eficiente. La ventaja de que no se modifiquen los objetos pasados como argumentos, propia del paso de los mismos por valor, se puede obtener ahora declarando el objeto como constante. La última elección entre puntero y referencia se salda, como es costumbre en C++, a favor de la referencia, que posee una sintaxis más clara y, en el fondo, físicamente, es un puntero constante a un objeto ya existente. El hecho, por otra parte, que el objeto cuya referencia se pasa como argumento se declare como *const* ayuda al compilador, pues éste ya sabe que el objeto no va a ser modificado, a optimizar la llamada.

Propongamos, sin más dilación, a efectos de examen, una posible implementación de esta sobrecarga:

[18] Y esto sin contar con la posibilidad que el objeto contenga otros objetos instanciaciones de clases y que, recursivamente, serían copiados mediante sus respectivos constructores de copia. No contamos, así mismo, con la posible construcción para su paso por valor de las correspondientes partes de las clases base de la clase dada.

```
Racional  Racional::operator+( const Racional& argDcho)
{
        numerador = numerador * argDcho.denominador
                        + denominador * argDcho.numerador;
        denominador = denominador * argDcho.denominador;
        return *this;       // devuelve el objeto que ha recibido
                                //  el mensaje, con su representación
                                //  interna convenientemente modificada
}
```

Esta definición supone que si codificamos lo siguiente:

```
Racional unMedio( 1, 2 ), dosTercios( 2, 3 );
Racional fraccionSuma = unMedio + dosTercios;
```

obtengamos un valor interno para fraccionSuma de '7/6', lo cual es correcto. El objeto dosTercios sigue representando el valor '2/3'. Pero, ¿qué pasa con el objeto unMedio? ¡Pues que su valor interno ha pasado a ser '7/6'! Repasemos la secuencia: en primer lugar la aplicación del operador suma equivale a la siguiente notación funcional:

```
unMedio.operator+( dosTercios );
```

que, asimilada a la implementación de la función detallada más arriba, resulta en que los datos miembros del objeto a través del que se llama la función (al que va dirigido el mensaje del operador: esto es, el objeto unMedio) son modificados y trocados por los resultantes de la suma algebráica de los racionales matemáticos representados por los objetos. O sea, que los datos originales del objeto unMedio se pierden para siempre. Seguidamente, la aplicación del operador devuelve el mismo objeto unMedio ya modificado, que es usado por el constructor de copia implícito para la construcción del objeto fraccionSuma. Bueno, la verdad es que esto dista bastante de lo que queremos conseguir con la operación suma. Intentemos, pues, una implementación distinta:

```
Racional Racional::operator+( const Racional& argDcho )
{
        Racional temp(      numerador * argDcho.denominador +
                            denominador * argDcho.numerador,
                            denominador * argDcho.denominador );
        return temp;
}
```

Esto es otra cosa: ahora se crea un objeto temporal de tipo *Racional*, distinto de los sumandos, utilizando el constructor de la clase con los argumentos del resultado de la suma. Si volviéramos al código de aplicación inmediatamente anterior veríamos que el objeto fraccionSuma resulta en valer '7/6', mientras que el objeto unMedio no ha cambiado su representación interna. Pero, ¿por qué devolver un objeto *Racional* en lugar de una referencia a un objeto *Racional*? ¿No parece esto último, de acuerdo con lo expuesto anteriormente, mucho más eficiente? Bien, es una buena pregunta. Intentaré responderla. Si cambiamos el tipo de retorno a *Racional&*, el código anterior podría quedar, salvo eso, invariante: al ejecutar la última línea de la función se devolvería una referencia al objeto temp. Pero este último objeto es local al cuerpo de la función, por lo que al devolver una referencia al mismo estamos preludiando el desastre: tras ejecutar la sentencia de retorno, devolviendo en este hipotético caso una referencia, el objeto temp será desinicializado por el destructor de su clase, de forma que la referencia, antes siquiera que podamos utilizarla, "apuntará" a un objeto que ya no

existe. ¡Diantre! -podría exclamar aquí el lector-, pero ¿el objeto local temp no se destruye, de cualquier manera, al salir del ámbito de la función de operador, independientemente de si tal función devuelve un objeto o una referencia, o aun un puntero? ¡Efectivamente! Pero únicamente cuando se devuelve un objeto lo que se devuelve no es el objeto local, sino una copia[19] del mismo (antes, evidentemente, de ser destruido), que se enmarcará en el ámbito al que pertenece la expresión de llamada a la función.

De acuerdo, de acuerdo -pensará el lector-, pero ¿y si creamos el objeto temp y lo alojamos en el área de almacenamiento libre con un código parecido al siguiente?

```
Racional& Racional::operator+( const Racional& argDcho )
{
        Racional temp = new Racional(
                                numerador * argDcho.denominador +
                                denominador * argDcho.numerador,
                                denominador * argDcho.denominador );
        return temp;
}
```

Así solucionamos el problema anterior, pero de hecho salimos de un pozo para caer en otro más profundo. Ahora devolvemos una referencia a un objeto creado mediante el operador *new*. Esto quiere decir que deberá ser expresamente destruido mediante el operador *delete*, pero normalmente esto es poco menos que imposible. Y si no examinen el siguiente código:

```
Racional cero, uno( 1 ), dos( 2, 1 );
Racional suma = cero + uno + dos;
```

Aquí uno se suma a cero, devolviendo una referencia a un objeto *Racional* temporal, al que se sumará el objeto dos, devolviendo una referencia a un nuevo objeto también temporal, con el que se construirá el objeto suma. Se han creado, pues, dos objetos temporales mediante el operador *new*, pero lo cierto es que por ser innominados no podemos acceder a ellos (a no ser que, con una tozudez cercana a la de Icaro, despachemos, extendidas por todo el código, declaraciones y asignaciones de identificadores que permitan nominar y tomar las direcciones de estos objetos temporales) , por lo que, en consecuencia, no podremos aplicarles el operador *delete*, lo que originará que, poco a poco, ineludiblemente, contruibuiremos a gastar la memoria disponible por la aplicación, hasta el punto de poder llegar al fatídico mensaje de falta de memoria. Vemos, pues, que la mejor solución es la originalmente propuesta de devolución de un objeto en lugar de una referencia.

LOS PELIGROS DE LAS SOBRECARGAS

Quizá habría que hablar, más bien, de los peligros de la inteligencia: los desarrolladores, torturados por un darwiniano afan de explicitar su brillantez, suelen proyectar en su código su personal identidad y, si cabe, su ingenio. Y si en programación estructurada esto es discutible, en el diseño de clases se convierte, la mayoría de las veces, en inaceptable, dando lugar,

[19] Como ya se ha señalado repetidas veces en el texto, el paso por valor o copia de un objeto se realiza mediante la aplicación del *constructor de copia*, bien implícito bien expreso, con el que cuentan todas las clases, como también ha quedado suficientemente explicado.

básicamente, a una deformación que podemos nominar como *sobrecarga no-intuitiva*. Veámoslo con más detalle, para lo que montaré un pequeño tour sobre el tema.

Como ya sabemos, no podemos "inventar" nuevos operadores: únicamente podemos sobrecargar los existentes. ¿Todos los existentes? ¡No! Los siguientes operadores no pueden ser sobrecargados:

```
::      .      .*
```

¿Por qué? Pues porque ya tienen un significado preciso y, tras conocer los mecanismos del lenguaje, *intuitivo* para el lector: tales operadores, entre todos los demás, se aplican ya, con carácter predefinido, sobre clases y sus instanciaciones. No se pueden sobrecargar, así mismo, los siguientes operadores:

```
?:      #      ##
```

pues ya poseen, también, un significado muy preciso: una redefinición de los mismos únicamente podría causar confusión y tristeza en el lector. Vemos, pues, que el lenguaje C++ tiende a evitar, aun en detrimento de la flexibilidad, las situaciones susceptibles de generar una arbitraria apariencia *no-intuitiva* del código.

En síntesis, se pueden sobrecargar los siguientes operadores:

```
+       -       *       /       %       ^       &       |       ~       ->*     =
<       >       +=      -=      *=      /=      %=      ^=      &=      |=      <<
>>      >>==    <<==    ==      !=      <=      >=      &&      ||      ,       ->
!       ++      --      ()      []      new     delete
```

y, de éstos, los siguientes como unarios:

```
+       -       *       &
```

Las sobrecargas, por otro lado, respetarán la signatura (binaria o unaria) de los operadores y mantendrán invariante el orden de precedencia de su evaluación, así como la dirección de su asociatividad. O sea, se respetará la tabla que se detalla a continuación, donde el lector agradecerá la inclusión de los operadores propios de C++ en la conocida relación de operadores de C, y en donde la dirección de la asociatividad está reflejada en la columna encabezada por 'A', de forma que 'D' significa "asociatividad por la derecha", mientras que 'I' equivale a "asociatividad por la izquierda". El orden de precedencia de los operadores viene dado por el orden del bloque en la tabla a que el operador se adscriba. Así el primer bloque, formado por el operador ::, es el de más alta precedencia, mientras que el operador coma es el de más baja precedencia. No existen precedencias de evaluación, empero, respecto de los operadores adscritos a un mismo bloque.

Los manuales de estilo de C++ enfatizan, también, la incoveniencia de sobrecargar los siguientes operadores:

, || &&

Operadores	Descripción	A
:: ::	cualificador de acceso a ámbito global cualificador de acceso a ámbito clase	D I
->, . () []	selectores de miembros de clases llamada a función o constructor índice de arrays	I I I
~ ++, -- sizeof ! +, - * () new, delete &	complemento a cero de bits autoincremento, autodecremento tamaño en bytes negación lógica suma y resta unarias desreferenciación conversión de tipo (cast: moldeo) alocación de memoria de almac. libre dirección-de	D D D D D D D D D
->*, .*	selectores de punteros miembros clases	I
*, /, %	relaciones multiplicativas	I
+, -	operadores aritméticos binarios	I
<<, >>	desplazamiento de bits	I
<, <=, >, >=	operadores relacionales	I
==, !=	igualdad, inegualdad	I
&	operador AND sobre bits	I
^	operador XOR sobre bits	I
|	operador OR sobre bits	I
&&	operador AND lógico	I
||	operador OR lógico	I
?:	operador condicional aritmético	I
=, *=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=	operadores de asignación	D
,	operador coma	I

pues, en caso contrario, se perdería la cualidad de secuenciación inherente a los mismos.

Como podemos apreciar, permanentemente se refuerza la idea de sobrecargas *naturales*, que no modifiquen la operatividad *intuitivamente* esperada de los operadores. Pero, cuidado, a veces lo que creemos más intuitivo nos puede llevar a extrañas situaciones. Pensemos en nuestra clase *Racional*: ¿por qué no dotarla con un operador de exponenciación? Y, para este fin, ¿qué mejor elección que la del operador '**^**', al que ya estamos acostumbrados por otros lenguajes? Instalemos, a modo de prueba, esta opción:

```
class Racional {
public:
        Racional operator^( const Racional& );
        Racional operator+( const Racional& );
        // sigue descripción de clase
};
```

Ahora podríamos codificar líneas como las siguientes:

```
Racional resultado, uno( 1 ), dos( 2 ), tres( 3 );
resultado = uno ^ dos + tres;
```

¿Cuál es el valor interno del objeto resultado? Si aplicamos las reglas de precedencia de operaciones que hemos aprendido en el grado escolar elemental, donde la exponenciación precede a la suma, tendremos que resultado contiene la fracción '4/1'. ¿Correcto? ¡No! ¡Estamos en C++, y la precedencia (y la asociatividad, en su caso) viene dada por la tabla anterior! Y en tal tabla podemos apreciar que la precedencia del operador '^' está varios niveles por debajo del la del operador '+'. ¿Qué ocurre, entonces, con nuestra expresión? ¡Pues que resultado contendrá la fracción '1/1'! O sea, el uso del operador más *intuitivo* para una operación nos ha llevado a un *antinatural* orden de evaluación de la misma. ¿Conclusión? Debemos evitar esta sobrecarga. ¡No veo por qué! -podría afirmar un lector-: sólo bastaría con explicitar la precedencia deseada de la siguiente forma:

```
resultado = ( uno ^dos ) + tres;
```

De acuerdo: esto soluciona el problema pero, a la vez, obliga al cliente de la clase al permanente uso de paréntesis en su código, lo que inadvertidamente podría olvidarse o calificarse como superfluo conduciendo a errores del tipo del expuesto. Tanto es así que la comunidad C++ expresamente desaconseja el uso de este operador para significar la exponenciación. ¿Cuál operador, entonces? Bueno, esto todavía está discutiéndose: recientemente he recibido del comité ANSI X3J16 C++ el texto de una propuesta relacionada con la adopción por el estándar del lenguaje de un operador autónomo de exponenciación[20]. O sea: no somos nadie. Intentando ser

[20] Este documento de Matthew H. Austern, titulado "Una propuesta para añadir un operador de exponenciación al lenguaje C++", X3J16/92-0099, WG21/N0176, propone la adopción por el lenguaje de dos nuevos operadores estándar: ***^** y ***^=**, donde el primero sería un operador binario de exponenciación y el segundo uno del conocido tipo *operador*=, con un nivel específico de precedencia y pudiendo ser normalmente sobrecargados.

prudentes, nuestra actuación deberá ser ecléctica, ponderando el uso de funciones expresas de la forma *elevadoA(...)* o *pow(...)*.

¿Qué pasa, por otra parte, con los operadores unarios? Pues simplemente que la función operadora miembro correspondiente no dispondrá de argumentos (recordemos que el objeto que llama a tal función sería su único operando). El problema, por llamarlo de alguna forma, podría aparecer con los operadores de autoincremento y autodecremento (++, --) porque, en su ámbito de actuación predefinido, operarán de forma distinta según prefijen o postfijen a su operando, y debemos encontrar alguna manera de expresar esta característica al sobrecargarlos. Realmente hasta AT&T C++ 2.1 no había posibilidad de diferenciar el uso prefijo o postijo de estos operadores: siempre operaban como prefijos. AT&T C++ 3.0 ha cambiado el panorama, aun a costa de un truco algo artificioso: el operador postfijo (para distinguirlo del prefijo) incorpora un argumento adicional de tipo *int*, ocupándose el compilador de proporcionarle un valor por defecto (que no nos interesa por su inutilidad práctica):

```
class Racional {
public:
        Racional& operator++();      // PREFIJO
        Racional operator++( int );  // POSTFIJO
        // sigue descripción de clase
};
```

Pero, entonces, ¿esto es una mera facilidad o se corresponde a la intención de que las sobrecargas prefijas y postfijas adecúen su comportamiento a la actuación predefinida de las mismas? Es decir, ¿la sobrecarga del operador postfijo ++ en nuestra clase *Racional* deberá devolver exactamente el objeto *Racional* a que se aplique y después incrementarlo en una unidad? Bien, esto es lo más intuitivo, y parece que también lo más correcto. ¿Cómo se codificaría? Veámoslo:

```
Racional operator++( int )
{
        // usaremos el constructor de copia implícito
        Racional temp = *this;
        // ahora incrementamos en una unidad nuestro objeto
        numerador += denominador;
        // seguidamente devolvemos una copia de nuestro objeto
        //  antes de ser incrementado en la unidad
        return temp;
}
```

Como el lector podrá fácilmente comprender, teniendo en cuenta lo expuesto en parágrafos anteriores, los operadores prefijos tenderán a devolver referencias a objetos (el mismo objeto convenientemente modificado), mientras que los operadores postfijos tenderán a devolver copias de objetos (la correspondiente a nuestro objeto antes de ser modificado).

Resumiendo: la sobrecarga de operadores en C++ es una posibilidad, no una necesidad, y debe ser usada juiciosamente. Debe intentar preservarse, ante todo, el sentido *intuitivo* de la actuación de los operadores en las sobrecargas (sería digna del empalamiento transilvano la sobrecarga del operador + para que actuara como una resta aritmética). De cualquier forma, si comenzamos a sobrecargar operadores en una clase, deberemos definir un interfaz completo de sobrecargas, pues otra cosa confundiría al cliente de la clase. Imaginen que sobrecargamos el operador + en nuestra clase *Racional* y que, sin embargo, definimos una función denominada *resta* para implementar la sustracción: el usuario de la clase fácilmente pensaría que al diseñador se le practicó una lobotomía en algún incierto momento de su vida.

FUNCIONES AMIGAS DE UNA CLASE

Parece que se han sentado las bases para la descripción completa de nuestra clase, que quedaría, más o menos, de la siguiente guisa:

```
class Racional {
public:
        boolean operator==( const Racional& );
        boolean operator<( const Racional& );
        // …
        Racional operator+( const Racional& );
        Racional operator-( const Racional& );
        Racional operator*( const Racional& );
        // …
        Racional( int = 0, int = 1 );
        ~Racional();
private:
        int numerador, denominador;
};
```

Recabemos ahora en un aspecto distinto sobre nuestro constructor. Podríamos considerar que el constructor es, en realidad, una función de conversión de una pareja de enteros, y aun de un entero, en un objeto distinto de tipo *Racional*. Posiblemente la idea podrá ser mejor aprehendida con el siguiente código:

```
Racional suma, uno( 1 );
suma = uno + 7;
```

donde, como ya sabemos, la última línea equivale a:

```
suma = uno.operator+( 7 );
```

¡Pero esta función sólo admite un argumento de tipo *Racional*, y sin embargo se le ha pasado un entero! ¿Cómo responde el compilador?

Intentando convertir tal entero en un objeto de tipo *Racional*, tal y como requiere el argumento de la función. ¿Y cómo realizar tal conversión? Intentando "construir" el objeto a partir del número entero. ¿Y cómo ... ? ¡Usando del constructor de nuestra clase! En definitiva puede decirse que ocurre lo siguiente: primero se crea un objeto temporal de tipo *Racional* para seguidamente ser usado como argumento de la función operador:

```
Racional objetoTemporal( 7 ); // "construye" el racional  '7/1'
suma = uno.operator+( objetoTemporal );
```

El objeto suma pasará, así, a detentar un valor interno de '8/1'. Ahora, circunstancialmente, podemos apreciar la ventaja de haber declarado parámetros por defecto en nuestro constructor. Si no hubiera sido así no tendríamos forma de "convertir" un entero en un objeto *Racional*. Observamos, también, la conveniencia de haber significado por la unidad el segundo argumento por defecto, pues la representación interna del objeto construido a partir de un solo entero se corresponde así con el concepto matemático que subyace tras tal representación. Veamos, sin embargo, qué ocurre con la siguiente expresión:

```
int siete = 7;
Racional unMedio( 1, 2 ), suma;
suma = siete + unMedio;
```

¡Pues lo mismo que anteriormente! -podría pensar alguno-: como la suma es conmutativa, tanto da 'x+y' que 'y+x'; se construirá un *Racional* a partir del entero y etc., etc. ¡Error, error y otra vez error! Orientemos nuestro pensamiento hacia los objetos y sus relaciones. ¿Qué viene a decir, bajo este punto de vista, la última línea del código expuesto? Pues que el objeto uno envía el mensaje operator+ (*suma*) al objeto siete de tipo *int*, el cual dispondrá de un método para responderlo, que presumiblemente generará un nuevo objeto que enviará el mensaje operator= al objeto suma. Pero, ¡el objeto de tipo *int* no dispone de ningún tal método!, por lo que el compilador señalará como errónea la expresión. Veamos, no obstante, la explicación funcional: de parecida forma a lo ya visto, la última línea del código equivale a la siguiente:

```
suma = siete.operator+( unMedio );
```

Y es claro que un objeto de tipo *int* no soporta tal función. ¿La suma algebráica no es, pues, una operación conmutativa? Sin duda, pero no así la función *operator+* de nuestra clase, pues tal y como la hemos implementado el *sumando* a la derecha del operador debe, por fuerza, ser un objeto ya construido de tipo *Racional*, por lo que no ha lugar a conversión posible alguna. Y, bien, ¿aquí se acaban nuestras alegrías? No, ciertamente. Sólo debemos reconsiderar nuestro planteamiento del esquema. Dado que el problema viene dado por el hecho que la función *operator+* es una función miembro de una clase, la solución más evidente radicaría en que tal función **no** fuera miembro de ninguna clase. ¿Qué tendríamos así? Pues una función global, cuya declaración, en nuestro caso, revistiría la siguiente apariencia:

```
Racional operator+( const Racional&, const Racional& );
```

De esta forma, cuando el compilador se encuentre con el código anterior (la suma infija de un entero y un *Racional*), aplicará la función global,

convirtiendo primero el entero en *Racional*. Entonces, ¿debemos codificar una función miembro y otra global para cada operador? ¿Se producirían, en este caso, errores por ambigüedad en la aplicación, para una determinada expresión, de una u otra función? Bien, vayamos por partes. Si definiéramos, por ejemplo, el operador + como función miembro y, a la vez, como función global, con sus respectivos argumentos de tipo *const Racional&*, veríamos que el código compila sin problemas, pues son funciones distintas (ni siquiera se trata de una sobrecarga). Pero -podría pensar el lector-, ¿como se decidiría qué función utilizar si se trata, por ejemplo, de una suma de *Racionales*, algo con lo que las dos funciones parece que encajan bien? De acuerdo: esta es una pregunta interesante. Veamos cómo reaccionaría el compilador ante el siguiente código:

```
int varint;
Prueba uno;
const Prueba dos,tres;
dos + tres;                    // función global
dos + uno;                     // función global
uno + tres;                    // función miembro
uno + dos;                     // función miembro
uno.operator+(dos);      // función miembro
uno + varint;            // función miembro
varint + dos;            // función global
uno + 1;                 // función miembro
2 + dos;                 // función global
```

Quizá el lector se pregunte: ¿no hay ambigüedades? ¿cuál es el método de resolución empleado por el compilador? Emprendamos un rápido tour por los recovecos del lenguaje.

Cuando el compilador se encuentra con una expresión del tipo a @ b, donde @ es un operador y donde bien a, bien b, bien ambos a y b son objetos instancias de clases, hay tres tipos de funciones a ser consideradas:

- operadores predefinidos @, aplicados sobre tipos predefinidos y/o instancias de clases susceptibles de ser convertidas a tipos predefinidos, implícita o explícitamente.
- funciones miembros del tipo *operator@(...)*, cuando a sea una instancia de clase, pertenecientes a la clase a cuyo tipo a corresponde.
- funciones no-miembros del tipo *operator@(...)*.

las cuales, a pesar de lo que pudiera pensar el lector, observan la misma precedencia a efectos del procedimiento de resolución que estamos detallando.

Seguidamente entra en escena el siguiente esquema secuencial, conocido como la **regla de intersección**:

- se desechan las funciones que no puedan ser usadas, por su número o tipo de argumentos, para resolver la llamada.

- si no existe al menos una función que "encaje", se produce un error y termina la secuencia.
- metafóricamente hablando, se abren dos especie de "bolsas", futuras contenedoras de funciones que puedan encajar con nuestra expresión: una que llamaremos "bolsaIzquierda" y otra "bolsaDerecha".
- se buscan las funciones que *mejor encajen* con el primer argumento (atención: no las que encajen, sino las que *mejor* lo hagan), y el conjunto de ellas se introduce en lo que habíamos llamado "bolsaIzquierda".
- se buscan las funciones que *mejor encajen* con el segundo argumento, insertando el conjutno de ellas en la denominada "bolsaDerecha".
- se crea una nueva "bolsa", que denominaremos "bolsaInterseccion", representativa del conjunto *intersección* de la "bolsaIzquierda" y la "bolsaDerecha".
- si la "bolsaInterseccion" contiene más de una función, la llamada será calificada como ambigua, y la secuencia acabará.
- si la "bolsaInterseccion" contiene únicamente una función, y esta función *encaja mejor*, en al menos un argumento de nuestra expresión, que el resto de las funciones de las bolsas "bolsaDerecha" y "bolsaIzquierda", entonces la llamada se resolverá a favor de esta función.
- en caso contrario (o sea, si la función encontrada no es la *mejor función de encaje*), entonces se produce un error por ambiguedad en la llamada.

Bien, pero ¿cómo se determina cuándo una determinada función se corresponde con uno de los *mejores encajes (best matchings)* posibles? Pues aplicando las cinco reglas de resolución que tuvimos ocasión de revisar cuando avistamos la sobrecarga de funciones, detalladas en la sección 13.2 de ARM : si el "encaje" de una función se ajusta a una regla con precedencia sobre la que se ajustaría el "encaje" de otra función, aquélla primera será un *mejor encaje* que esta última.

Antes de revisar el ejemplo propuesto en este mismo parágrafo, debemos indicar que, de acuerdo con lo establecido en la sección 13.2 de ARM, páginas 316-317, para propósitos de *encaje*, una función miembro no-estática se considerará como una función global con un argumento extra, el cual deberá *encajar* con el primer operando del operador sobrecargado y que será bien del tipo *a&* (para objetos instanciaciones de la clase *a*) en su caso más general, bien de los tipos *const a&* o *volatile a&* para funciones miembros *const* y *volatile* repectivamente. Bueno, esto puede parecer confuso, así que lo mejor será que lo veamos en la práctica, repasando bajo estas nuevas luces el ejemplo anterior.

Tenemos, en principio, unas cuantas líneas de código con operadores infijos de suma, por lo que, desechando las demás funciones, encontramos dos prototipos que podrían encajar con tales expresiones:

```
// función global
Racional operator+( const Racional&, const Racional& );
clase Racional {
public:
        // función miembro
        Racional operator+( const Racional& );
};
```

donde, únicamente a efectos de *encaje (match)*, la función miembro, con arreglo a la regla expuesta, equivale a la siguiente función global:

```
Racional operator+( Racional&, const Racional& );
```

Examinemos ahora la primera expresión, en la que se suman dos objetos constantes de tipo *Racional*. ¿Cuál de estas dos funciones encaja mejor con respecto al primer operando? Como el segundo argumento de nuestras funciones es el mismo, únicamente nos deberemos preocupar del primer argumento. Al ser el primer operando un objeto constante, el *mejor encaje* corresponderá a la función global (que meteremos en la bolsaIzquierda). ¿Qué función encaja mejor, seguidamente, desde el punto de vista del segundo operando? Ambas funciones, pues en ambas el segundo argumento es una referencia a un objeto *Racional* constante, como constante es el segundo operando (meteremos en la bolsaDerecha las dos funciones). ¿Qué contendrá la bolsa Intersección? Una única función: la global. El resultado coincide, pues, con el expuesto en el ejemplo.

Veamos otra expresión: la que suma un objeto *Racional* a un entero situado a la derecha del operador. Con respecto al primer operando, el mejor *encaje* lo proporciona la función miembro, pues, siendo el objeto no constante, se prefiere un ajuste exacto a una conversión trivial de *Racional&* a *const Racional&*, tal y como requeriría la función global. Con respecto al segundo operando, ambas funciones poseen el mismo rango de *mejor encaje*. ¿La intersección? Únicamente la función miembro.

Vemos, pues, que al diferir nuestras funciones únicamente en el "primer" argumento, si el primer operando de la expresión no es constante la mejor opción será la función miembro, y la función global si es constante. Por supuesto la mejor opción será siempre la función global en el caso de un *int* (o *double* o *float*) como primer operando. Sería un buen ejercicio para el lector la comprobación expresa de todas las expresiones.

¿Qué pasaría, sin embargo, si calificáramos nuestra función miembro como *const*? Pues que, a efectos de *encaje*, el primer argumento de la función habría pasado a ser *const Racional&*, con lo que al compilar el código obtendríamos seis errores por ambigüedad (todas las expresiones de suma a

excepción de las que disponen de un primer operando predefinido). El lector puede experimentar, igualmente, con los errores que aparecerían si, por ejemplo, los argumentos de la función global (el primero, el segundo o ambos) dejaran de ser constantes.

Pero no perdamos el hilo del relato: toda esta explicación ha venido a cuento de la necesidad de definir una función global para evitar algunas de las incoveniencias de las funciones miembros operadores. Notamos que pueden coexistir ambas versiones, pero que esto no tiene razón de ser y, si no afinamos mucho, puede conducirnos a problemas de ambigüedad. ¿Qué hacemos entonces? Pues suprimir la función miembro y dejar únicamente la función global, naturalmente.

¿Hemos solucionado todos nuestros problemas? Es posible. Abordemos un intento de definición de nuestra función global:

```
Racional operator+( const Racional& lhs, const Racional& rhs )
{
        Racional temp(      lhs.numerador * rhs.denominador +
                                lhs.denominador * rhs.numerador,
                                lhs.denominador * rhs.denominador );
        return temp;
}
```

Bueno, en apariencia parece suficiente. ¿Es correcto, pues? ¡No! Y el lector, a estas alturas, ya debería conocer la razón. El algoritmo de suma de racionales es, en esencia, correcto, pero examinemos detenidamente el código del cuerpo de la función: se construye un objeto temporal usando del constructor de dos argumentos de la clase, para lo que se opera algebráicamente con los datos internos (numerador y denominador) de cada uno de los objetos, accedidos a través de la notación '.' (*dot*). ¿Y bien? ¡Pues que un objeto no puede acceder directamente a sus miembros privados (como lo son *numerador* y *denominador*)! El compilador flagelará como erróneo el anterior código. ¿Qué hacer, entonces, para salvar esta nueva traba? Siguiendo el esquema de ocultación de la información, lo más evidente sería dotar a nuestra clase de sendas funciones miembros (posiblemente *inline*) públicas que "accedieran" a los datos privados. Probemos:

```
class Racional {
public:
        int num() const { return numerador; }
        int denom() const { return denominador; }
private:
        int numerador, denominador;
        // sigue resto descripción clase
};
```

De esta manera nuestra función global podría ser re-escrita de la siguiente (correcta) forma:

```
Racional operator+( const Racional& lhs, const Racional& rhs )
{
        Racional temp(      lhs.num() * rhs.denom()
                                + lhs.denom() * rhs.num(),
                                lhs.denom() * rhs.denom() );
        return temp;
}
```

Así que ¿hemos llegado, por fin, al desenlace de nuestros problemas? Bueno, lo normal es que, efectivamente, aquí acabe el proceso. Pero reexaminemos lo hecho: hemos creado unas funciones de acceso convenientes para definir nuestra función global, pero al declararlas en la sección *public* de nuestra clase cualquier cliente podría acceder a ellas. Preguntémonos: ¿es aceptable que cualquier cliente acceda y opere por separado con el numerador y el denominador de nuestros objetos *Racionales*? Coherentemente, no. Si pensamos en la conceptualización de tales objetos podremos apreciar que vienen definidos por la relación entre numerador y denominador, y no por los valores concretos de éstos. De hecho, como es bien conocido, una fracción (que podemos denominar *reducida* o *simplificada*) representa una serie infinita de fracciones múltiplos de la misma. Vemos, pues, que los valores concretos del numerador y del denominador no son, en absoluto, relevantes. Es más, podría resultar incluso peligroso permitir que ciertos cálculos del cliente se basaran en tales volátiles valores. Podríamos, incluso, sostener la conveniencia de una función interna (de acceso *private*):

```
class Racional {
private:
        Racional& simplificar();
        // sigue descripción de clase
};
```

de tal forma que pudiera ser usada por constructores, operadores y, en general, funciones miembros, para ofrecer así siempre una representación interna adecuada de nuestros objetos. ¿En qué afecta esto a nuestra función global? En nada, pues esta nueva función es privada (no tiene sentido dejar tal herramienta en manos de los clientes de la clase, pues así podríamos cambiar su nombre posteriormente, o aun el esquema de representación de nuestros objetos), por lo que no podrá ser accedida desde la función global. Bueno, en un continuo vaiven, hemos vuelto a caer en el pozo de la confusión.

Recapitulemos: con una función miembro tenemos el problema de que ciertas expresiones no compilan como sería de esperar, mientras que con una función global nos encontramos con serias restricciones de acceso a la información interna de los objetos. Necesitamos lo mejor de ambos mundos. Y aquí es donde entran en escena las funciones **friend**s (*amigas*)

Un amigo es una persona con la que se comparten las intimidades de uno. De igual forma una función amiga (*friend*) es, sencillamente, una función global con acceso a los miembros privados de una o más clases. ¿De qué clases? Pues de las clases amigas de la misma. ¿Y cómo ...? Bueno, para que una determinada función se "amigue" con una o más clases, lo único que hay que hacer es declarar tal función dentro del protocolo de descripción de estas clases antecedida por la palabra clave *friend*, como en el siguiente ejemplo:

```
class Racional {
        friend Racional operator+(    const Racional&,
                                                        const Racional& );
        friend boolean operator<=( const Racional&,
                                                        const Racional& );
        // sigue resto descripción de clase
};
```

Naturalmente, según lo expuesto, una función sólo puede hacerse *amiga* de una clase en el momento de definir ésta, pues de otra forma se vulneraría el principio de ocultación de la información: cualquiera podría acceder a los datos internos de una clase declarando una función cualquiera como *amiga* de la misma. La palabra clave *friend* se usa sólo en la declaración de la función, pudiendo ésta ser definida con normalidad:

```
Racional operator+( const Racional& lhs, const Racional& rhs )
{
        Racional temp(      lhs.numerador * rhs.denominador +
                                lhs.denominador * rhs.numerador,
                                lhs.denominador * rhs.denominador );
        return temp.simplificar();
}
```

Vemos que el código de construcción del objeto temporal es idéntico al de nuestro primer intento de función operador global, con la salvedad que esta función, por el hecho de haber sido declarada *amiga* de la clase *Racional*, puede acceder con total impunidad a la sección *private* de la misma y a los datos y funciones miembros que ésta contenga. Podríamos visualizar gráficamente la situación pensando en que una relación de *amistad* "abre" un agujero en el esquema de protección de una clase procurando que, a efectos prácticos y de acceso, la función se considere como un particular y heterodoxo "miembro" de tal clase.

Incidentalmente podemos notar que en el anterior código hemos hecho uso de la función miembro simplificar() (también *private*, pero ahora accesible por nuestra función *friend*), de manera que lo que ocurre en la última línea es lo siguiente: al objeto temporal construido como suma de los dos operandos de la función *friend* se le aplica un método de simplificación algebráica de la fracción interna que lo representa (diviendo numerador y denominador por el máximo común divisor de ambos), método que, como ya vimos, devuelve una referencia al objeto ya simplificado. La sentencia de retorno, por último, devuelve una copia de tal objeto al ámbito de llamada del operador. Por supuesto, esta función de simplificación habría de ser usada, también, en el cuerpo del constructor de la clase *Racional*, a fin de que, desde el principio, los objetos contengan su representación interna óptima.

¿En qué sección de una clase se declara una función *friend*? La verdad es que se puede declarar en cualquiera: no olvidemos que las secciones de acceso (controladas, como ya sabemos, por las etiquetas *public*, *private* y *protected*) únicamente identifican el nivel de protección de los miembros de la clase que contienen, y nuestra función *friend*, por no ser una función miembro, no puede ser calificada con tal criterio. Lo habitual es, de cualquier forma, declarar el bloque de funciones *friend* justo al principio de descripción de la clase, justo antes de declarar función o dato miembro alguno.

Bien, ya hemos resuelto en gran medida nuestro problema con los operadores infijos, pero debemos pensar lo siguiente: el cuidadoso esquema de protección de las clases, tras el normalmente costoso proceso de diseño que conlleva, es inmisericordemente vulnerado por las funciones *friend*s, y esto no parece muy lógico. Y, de hecho, no lo es. El lector deberá seguir **siempre** el siguiente esquema: intentar primero una función miembro, después una función global, y por último, si lo anterior no es posible, una función *friend*. Demasiadas funciones *friend* en una clase denotan bien una cierta pobreza en su diseño bien una lectura desquiciada de ciertos textos hindúes.

SOBRECARGA DE LOS OPERADORES DE INSERCIÓN Y EXTRACCIÓN

¿Qué métodos deberemos usar para las operaciones de entrada/salida de nuestros objetos racionales? Siendo coherentes, habremos de sobrecargar los operadores '<<' y '>>'. Veámoslo con el operador de inserción:

```
#include <iostream.h>          /* para poder declarar más
                                        adelante los tipos xstream */

class Racional {
public:
        ostream& operator<<( ostream& );
        // ...
};
```

O sea, el operador '<<', aplicado a uno de nuestros objetos racionales y a un *ostream (output-stream)*, tras ejecutar un código que simplemente imprimirá los datos miembros de los objetos en bonita apariencia, devolverá una referencia a un *ostream* (presumiblemente al objeto predefinido *cout*), de forma que la expresión envuelta pueda encadenarse y al *ostream* retornado puedan serle insertados nuevos objetos. ¿Correcto? Parece que hemos pasado por alto una circunstancia que pronto nos será muy familiar. ¿Cómo codificaríamos una expresión con tal operador? Pues nuestro objeto a la izquierda, tras él el operador, y seguidamente el *stream*. Algo así como:

```
Racional miRacional( 1, 2 );
miRacional << cout;              // horror: el mundo al revés
```

¿Qué es esto? No es la sintaxis normal a que estamos acostumbrados con los *streams*. No es tampoco, sin embargo, una codificación errónea: únicamente es *no-intuitiva*:

```
miRacional << cout << " es mi número\n";          // 1/2 es mi número
miRacional << ( cout << "Mi número es " );// Mi número es 1/2
```

Recordemos ahora lo dicho anteriormente sobre el operador de exponenciación: en C++ las expresiones *no-intuitivas,* o que deban ser cuidadosamente pensadas antes de escribirlas, han de evitarse como la peste bubónica. Tenemos aquí, pues, una razón, distinta a la de los operadores algebráicos, para la re-escritura de nuestra función como no-miembro de la clase, de forma que pueda ser variado el orden de los operandos.

Siguiendo la regla antes expuesta, primero debemos intentar una función global. Apreciamos, empero, que la más pausible implementación

```
ostream& operator<<( ostream& os, const Racional& fraccion )
{
        os      << fraccion.numerador << '/'
                << fraccion.denominador << '\n';
        return os;
}
```

vulnera el esquema de protección al intentar acceder, erróneamente, a los datos privados del objeto. La única solución sería, pues, declarar esta sobrecarga (y la del operador de extracción) como *friend*s:

```
class Racional {
        friend ostream& operator<<( ostream&, const Racional& );
        friend istream& operator>>( istream&, const Racional& );
        // sigue resto descripción clase
};
```

A partir de ahora nuestros objetos de tipo *Racional* pueden ser insertados con naturalidad en una cadena de impresión de objetos de otros tipos, o bien pueden ser extraídos de un *istream* con la misma facilidad sintáctica que un *float* o un *char*.

OPERADORES DE CONVERSIÓN

Ya hemos visto que, merced al constructor y a través de funciones miembros, globales y *friends*, se genera una suerte de "conversión" desde tipos predefinidos a el tipo *Racional*. De esta manera, en la práctica, podemos usar un *long*, un *int* o un *float* en cualquier expresión dondequiera que es esperado un *Racional*. Lo que parece perfecto, pero, ¿puede hacerse

a la inversa? O sea, ¿podemos intrumentar un procedimiento que permita usar un *Racional* dondequiera que se espera un *long* o un *double*? En definitiva, buscamos "algo" que permita codificar:

```
Racional unoRacional( 1 );
float unoFloat = unoRacional;                    // error
```

La solución es clara: solamente debemos dotar a la clase *Racional* de métodos de conversión para trocar el tipo de sus objetos, y esto se realizar mediante lo que se denominan operadores de conversión. Adelantaré, para avanzar rápido, un ejempo:

```
class Racional {
public:
        operator float() {
                return float( numerador / denominador );
        }
        // sigue resto descripción de clase
};
```

Hemos definido una nueva función miembro de nuestra clase, con la especial sintaxis de los operadores, sin argumentos y sin tipo alguno de retorno, lo cual es lógico si pensamos que, como operador de conversión, habrá de convertir un objeto *Racional* devolviendo, al fin del proceso, un *float*, por lo que ya se anuncia el tipo de retorno. De esta manera ya podríamos compilar sin problemas el anterior código.

Con cierta coherencia, los operadores podrán aplicarse para la conversión a tipos predefinidos y clases. Esto es, contando con que, por ejemplo, hubiéramos ya declarado la clase *Complejo*, podría definirse un operador de conversión de un objeto *Racional* a un objeto *Complejo*:

```
#include <complex.h>

class Racional {
public:
        operator float();
        operator complex();
        // ...
};
```

Parece así que, con las sobrecargas de operadores por un lado y los operadores de conversión por otro, nuestros objetos *Racionales* se comportarán adecuadamente sea cual sea su posición o uso en una expresión, asemejándose en tal aspecto a los objetos de tipos predefinidos. Bueno, lo parece pero no lo es. Veamos qué ocurre cuando el compilador se encuentra algo parecido a lo siguiente:

```
Racional unMedio( 1, 2 );
2.5 + unMedio;
```

Recapacite el lector: ¿qué devuelve esta última línea? ¿un *float* o un *Racional*? ¿acaso compilará? Bien, recontemos nuestro arsenal: tenemos una función *friend* para el operador de suma, que devuelve un *Racional*, y tenemos también un operador de conversión que, trocando el *Racional* en *float* y sumándole el primer valor, causaría que la expresión devolviera un *float*. Para resolver esta situación el compilador utiliza la Regla de Intersección, que ya detallamos al comentar las funciones amigas: con respecto al primer argumento (un *float*), el mejor encaje lo proporciona el operador de suma predefinido (tras la aplicación del operador de conversión a *float*); con respecto al segundo argumento el mejor encaje es el de la función *friend* que sobrecarga el operador de suma. La intersección de ambos conjuntos es nulo, por lo que el compilador flagelará la expresión de suma como un error por ambigüedad.

Pero bueno, exclamará el hastiado lector, ¡en C++ tras cada montículo superado aparece una montaña mayor! ¿Quiere esto decir que no podemos usar de forma concurrente las sobrecargas de operadores y los operadores de conversión? No exactamente. Imaginemos que añadimos a nuestra lista de funciones amigas las siguientes:

```
class Racional {
        friend Racional operator+( float, const Racional& );
        friend Racional operator+( const Racional&, float );
        friend  Racional operator+(    const Racional&,
                                                    const Racional& );
        // …
public:
        operator float();
        // …
};
```

Ahora la aplicación de la Regla de Intersección a la anterior expresión de suma generará un resultado diferente. Los mejores encajes con respecto al primer operando son la función operator+(float,const Racional&) y el operador predefinido de suma de *float*s (gracias al operador de conversión a *float*). Los mejores encajes con respecto al segundo operando los constituyen la función operator+(float,const Racional&) y la función operator+(const Racional&,const Racional&). La intersección ahora es, pues, la función operator+(float,const Racional&), con lo que ya no se da la ambigüedad anterior. Pero, ¿y si quisiéramos que el *Racional* operara como un *float* en tal expresión? Únicamente deberíamos realizar un *cast* expreso de nuestro objeto:

```
float resultado = 2.5 + (float)unMedio;          // Ok: no hay
                                                          //  ambigüedades
```

¿Se han resuelto entonces todas las ambigüedades? El lector ya puede suponer que eso sería demasiada felicidad. Veamos qué ocurre si codificamos:

```
long prueba;
Racional dosTercios;
```

```
float suma = prueba + dosTercios;       // error: ¡AMBIGUO!
```

¿Qué ha pasado aquí? Es fácil de ver: en la resolución de la operación de suma mediante la aplicación de la Regla de Intersección, los mejores encajes con respecto al segundo operando siguen siendo las dos funciones *friend* del caso anterior, pero el mejor encaje con respecto al primer operando es... ¡el operador predefinido de suma para *long*s! La intersección es vacía y, por tanto, la llamada es ambigua. Por supuesto aquí caben varias posibles soluciones. En caso que quisiéramos que prevaleciera el uso del operador predefinido, codificaríamos lo siguiente:

```
float suma = int( prueba ) + dosTercios;
```

Podríamos, por otra parte, en caso de desear el uso de la sobrecarga de operadores, explicitar un *cast* a *float* para la variable *long*; o podríamos definir nuevas funciones *friend* en nuestra clase cubriendo todos los posibles casos de operadores entre *Racionales* y tipos predefinidos. Pero la verdad es que todo esto no parece demasiado apropiado. Se nos antoja que la combinación de sobrecargas de operadores y de operadores de conversión no ajusta como debiera. Bien, esto es un hecho y los desarrolladores de C++ deben aprender a vivir con él.

Caben, en principio, dos posibles planteamientos: bien suprimimos todas las sobrecargas de operadores y operamos con conversiones, bien prescindimos de las conversiones "peligrosas". Por mi parte, a pesar de las discutibles ventajas de la primera opción, decido, teniendo además en mente que los *Racionales* son un superconjunto de los enteros, mantener las sobrecargas. No es deseable, empero, perder la capacidad de usar los *Racionales* como tipos predefinidos. ¿Podemos llegar a una solución de compromiso? Bueno, podemos dotar a nuestra clase de funciones miembros expresas de conversión, del tipo *convierteAEntero()*, etc:

```
class Racional {
public:
        int comoInt() {
                return int( numerador / denominador );
        }
        int comoFloat() {
                return float( numerador / denominador );
        }
        // etc, etc.
};
```

De esta forma ahora podríamos codificar lo siguiente:

```
Racional dos( 2 ), tresQuintos( 3, 5 );
int alFinSuma = 3 + dos.comoInt();
float sumaDeFloats = tresQuintos.comoFloat() + 2.5;
```

Como casi siempre, lo óptimo prevalece sobre lo mejor en C++. Esta pequeña revisión, por otro lado, de algunas "sutilezas" del lenguaje espero haga comprender a los lectores la necesidad -imperiosa- de entender con cierta profundidad los mecanismos del mismo. Y para esto hay que leer, leer, leer. Y estudiar el ARM.

FUNCIONES MIEMBROS ESTÁTICAS

Como ya sabemos cada objeto de nuestra clase *Racional* posee una representación interna distinta, de forma que las funciones miembros, a través del puntero implícito *this*, quedan particularizadas y pueden acceder a unos datos concretos correspondientes al objeto desde el que se llaman. Esto es perfecto para unos datos internos que se suponen diferentes para cada objeto, pero ¿qué pasa si deseamos dotar a nuestros objetos de unos datos compartidos por la totalidad de los otros objetos de la clase? ¿o si queremos implementar en un objeto un método que no dependa de la representación interna del mismo? Bueno, ojeemos el horizonte.

Imaginemos que deseamos controlar el número de objetos de tipo *Racional* activos en un momento dado. Hablamos, en definitiva, de una variable de tipo *int* que podemos identificar como *numeroDeRacionales*. ¿Corresponde tal variable a la representación interna de un objeto concreto? Evidentemente no. ¿Se trata entonces de una variable global? Pudiera ser. De hecho si fuéramos consecuentes con el tradicional esquema de desarrollo estructurado ésta sería la representación idónea. Pero pensemos que tal variable únicamente tiene sentido para los objetos de nuestra clase, por lo que dotarla de ámbito global de programa es una acción similar a la del novio que se compra cinco trajes para la ceremonia nupcial. En realidad tal variable debería estar adscrita al ámbito de nuestra clase *Racional*, con lo que conseguiríamos dos cosas por el precio de una: en primer lugar evitar conflictos de nombres globales, y en segundo lugar poder aplicar el esquema de ocultación de la información a tal variable. Efectivamente no hay ninguna razón para suponer que la variable *numeroDeRacionales*, a pesar de poder ser compartida por todos los objetos, deba poseer acceso público, y más bien uno se da en pensar que en este caso la privacidad habría de ser preservada. Muy bien, ¿y como ... ? Pues declarando tal variable como un dato miembro estático (**static**) de nuestra clase:

```
class Racional {
private:
        int numerador, denominador;
        Racional& simplificar();
        static int numeroDeRacionales;
public:
        Racional( int = 0, int = 1 );
        // ...
};
```

A diferencia de lo que ocurre con el resto de datos miembros, la declaración de un dato miembro *static* no es una definición. Esto es, el desarrollador debe proveer una inicialización o definición expresa en algún lugar del código, preferentemente en el archivo en que se implementan las funciones de la clase. La inicialización, por otro lado, al igual que ocurre con las variables globales estáticas, únicamente puede realizarse una vez. Veamos cómo:

```
int Racional::numeroDeRacionales = 0;
```

Bueno, este código requiere algún comentario. En primer lugar vemos que es necesario aplicar a nuestra variable el operador de resolución de ámbito, lo cual es lógico, pues la visibilidad de ésta se limita al ámbito de la clase. Podríamos preguntarnos seguidamente: ¿qué pasa con el nivel de acceso? Esta variable es privada y, sin embargo la hemos inicializado con ámbito global de fichero. En realidad esta es una particularidad de los miembros estáticos de clases: el nivel de protección se refiere a las operaciones de acceso y modificación de los mismos, pero no afecta a su inicialización o definición.

¿Cómo podría usarse este nuevo miembro de la clase? Lo que parece más claro es que el constructor (o constructores en otros casos) incremente en una unidad el dato miembro estático cada vez que se cree un objeto, a la vez que el destructor minorará, igualmente, una unidad cada vez que se destruya uno.

```
Racional::Racional( int num, int denom )
        : numerador( num ), denominador( denom )
{
        simplificar();
        Racional::numeroDeRacionales += 1;
}

Racional::~Racional()
{
        Racional::numeroDeRacionales -= 1;
}
```

Constructor y destructor pueden acceder al dato miembro estático, a pesar de ser privado, precisamente por ser funciones miembros de la clase. Pero sigamos con el esquema de ocultación de la información: *numeroDeRacionales* es un dato miembro privado y, al igual que haríamos con otros miembros no-estáticos, debemos proveer a la clase al menos de una función de acceso a tal variable. Intentémoslo:

```
class Racional {
public:
        static numeroDeObjetosRacionales() {
                return Racional::numeroDeRacionales;
        }
```

```
        // ...
};
```

¿Es imprescindible que una función que acceda a un miembro estático sea también estática? ¡No! Lo que sí se cumple es, precisamente, lo contrario: una función miembro no podrá ser calificada como estática si accede al menos a un dato o función miembro no-estática. Hay que entender que los miembros estáticos no forman parte de los objetos: los datos miembros estáticos se constituyen en objetos diferenciados y aparte, y las funciones miembros estáticas existen y pueden ser llamadas aunque no se hubiera construido ningún objeto de la clase a que pertenecen. Por esta razón las funciones miembros estáticas no pueden acceder a los miembros de un objeto (a excepción de a los públicos, naturalmente), y para verlo claro el lector sólo debe preguntarse: ¿a los miembros de qué objeto? Pensemos que la llamada a funciones miembros no estáticas se realiza desde un determinado objeto. Igualmente desde un objeto cualquiera se puede acceder a una función miembro estática, pero esto únicamente se debe a una cuestión de coherencia sintáctica: de hecho el objeto desde el que se accede a tal función no es en absoluto importante (tanto es así que ni siquiera es evaluado por el compilador) y constituye lo que se llama una extensión *dummy* de completitud sintáctica. Las funciones miembros estáticas pueden accederse también, y esto es lo más normal y recomendable, directamente. Veámoslo:

```
// acceso directo
cout << Racional::numeroDeObjetosRacionales();
Racional cualquiera;
// función miembro
cout << cualquiera.numeroDeObjetosRacionales();
```

Como el lector probablemente ya habrá adivinado, el hecho que las funciones miembros estáticas no pertenezcan a ningún objeto fuerza a que no contengan al puntero implícito *this* (¿a qué objeto apuntaría?). Es más: cualquier invocación expresa de tal puntero implícito en su cuerpo sería calificada como un error. Lo más adecuado para el novicio sería pensar siempre en estas funciones como funciones globales estáticas, aunque con especiales niveles de visibilidad y acceso.

Los datos miembros estáticos, desde el momento de su declaración en el protocolo de descripción de una clase, pueden ser utilizados como valores en listas de parámetros por defecto, así como pueden, por el mismo hecho que su declaración no es una definición, ser ojetos de la clase de la que son miembros.

Naturalmente los miembros estáticos expuestos quizá no sean demasiado apropiados para incorporarlos a nuestra clase, pues, en definitiva, se han creado como ejemplificación pedagógica de la sección.

UN EPÍLOGO DE COLUSIÓN

¿Hemos completado ya el diseño de la clase *Racional*? Parece que así es: tras un incesante vaiven de problemas, dudas, indecisiones y sutilezas, el interfaz de nuestra clase parece haber tomado una forma bien definida. El lector podría pensar: "Ha sido ...(ourps!), pero las cosas han quedado medianamente claras al final". Pero pongamos al lector en un brete, detallando un planteamiento no visto: imaginemos que hacemos coexistir a funciones operadoras globales y miembros. ¿Cómo? He aquí un ejemplo:

```
// función global
Racional operator+( float, const Racional& );

class Racional {
public:
        Racional operator+( const Racional&, const Racional& );
        // ...
};

inline Racional operator+( float miFloat, const Racional& rhs )
{
        return rhs.operator+( miFloat );
}
```

De esta manera la función global, definida *inline* (con lo que no se ocasiona penalización en su llamada) soporta el único caso problemático que no puede tratar bien la función operadora miembro, implementando una suerte de conmutatividad forzada. Dejo como ejercicio al siempre amable lector la comprobación de la idoneidad o no de este planteamiento: su practicidad, si ocasiona o no errores por ambigüedad, su relación con los operadores de conversión, etc. Y es que en C++ hay que estar preparado para todo.

Otra sorpresita: ¿realmente interesa una representación interna de numerador y denominador como *int*s? Yo creo que no. Si realmente deseamos otorgar de operatividad práctica y flexibilidad a nuestra clase, una representación en *long*s sería más apropiada. Cambiemos, pues, estos tipos. ¿Mucho trabajo? ¡No! Piénsese que únicamente estamos completando el interfaz de la clase, por lo que los cambios afectan sólo a una pequeña porción de código: otra de las ventajas de separar interfaz e implementación.

INTERFAZ DE LA CLASE *RACIONAL*

Lo que sigue es un intento elemental de descripción de la clase Racional, teniendo en cuenta que se han suprimido grupos habituales de funciones miembros, como los referidos a operaciones de exponenciación, valor absoluto, etc., de fácil implementación por el lector.

```
#ifndef RACIONAL_H
```

```
#define RACIONAL_H

// fichero:                 RACIONAL.H
// contenido:       interfaz de la clase "Racional",
//                          representacion del conjunto matematico
//                          de los numeros racionales.
// autor:           Ricardo Devis
// derechos:        (C) INFO+ (1993)
//                          Este codigo puede copiarse libremente,
//                          con la unica restriccion de respetar
//                          el aviso de copyright.
// last mod:        03-06-93

#include <iostream.h>

typedef int boolean;

class Racional {

        // operadores de entrada/salida
        friend ostream& operator<<(             ostream& os,
                                                const Racional& );
        friend istream& operator>>(  istream& is, Racional& );
        // operadores artimeticos
        friend Racional operator+(    const Racional&,
                                                const Racional& );
        friend Racional operator-(    const Racional&,
                                                const Racional& );
        friend Racional operator*(    const Racional&,
                                                const Racional& );
        friend Racional operator/(    const Racional&,
                                                const Racional& );
        // operadores de comparacion
        friend boolean operator==(   const Racional&,
                                                const Racional& );
        friend boolean operator!=(    const Racional&,
                                                const Racional& );
        friend boolean operator>(     const Racional&,
                                                const Racional& );
        friend boolean operator<(     const Racional&,
                                                const Racional& );
        friend boolean operator<=(   const Racional&,
                                                const Racional& );
        friend boolean operator>=(   const Racional&,
                                                const Racional& );

public:
        // constructor y destructor inline
        Racional( long num = 0, long denom = 1 )
                : numerador( num ), denominador( denom )
        {
                simplificar();
        }
        ~Racional() {};
```

```
        // operadores unarios
        Racional operator-();
        Racional& operator+();
        // operadores prefijos
        Racional& operator++();
        Racional& operator--();
        // operadores postfijos
        Racional operator++( int );
        Racional operator--( int );
        // operadores logicos
        boolean operator!();
        // operadores de asignacion
        Racional& operator+=( const Racional& );
        Racional& operator-=( const Racional& );
        Racional& operator*=( const Racional& );
        Racional& operator/=( const Racional& );
        // funciones de conversion
        float comoFloat();
        long comoLong();
        double comoDouble();
        int comoInt();

private:
        long numerador;
        long denominador;
        // funcion interna para simplificar fracciones
        Racional& simplificar();
};


#endif  /* RACIONAL_H */
```

IMPLEMENTACIÓN DE LA CLASE *RACIONAL*

A excepción del constructor y el destructor (lastrados por un interés eminentemente pedagógico), todas las demás funciones se definen en el siguiente módulo, incluso las inline. Note el lector que funciones como por ejemplo el operador '!=' se aprovechan de la anterior definición de su operador complementario ('==') y de la característica de inlining, trocando el código más intuitivo.

```
// fichero:             RACIONAL.CPP
// contenido:   implementacion de la clase "Racional",
//                      representacion del conjunto matematico
//                      de los numeros racionales.
// autor:       Ricardo Devis
// derechos:    (C) INFO+ (1993)
//                      Este codigo puede copiarse libremente,
//                      con la unica restriccion de respetar
//                      el aviso de copyright.
// last mod:    03-06-93
```

```
#include "racional.h"

ostream& operator<<( ostream& os, const Racional& rhs )
{
        os << rhs.numerador << '/' << rhs.denominador;
        return os;
}

istream& operator>>( istream& is, Racional& rhs )
{
        char div;
        long num, denom;
        is >> num >> div >> denom;
        if ( is )
                rhs = Racional( num, denom );
        return is;
}

Racional operator+( const Racional& lhs, const Racional& rhs )
{
        Racional temp(  lhs.numerador * rhs.denominador
                        + lhs.denominador * rhs.numerador,
                        lhs.denominador * rhs.denominador );
        return temp.simplificar();
}

Racional operator-( const Racional& lhs, const Racional& rhs )
{
        Racional temp(  lhs.numerador * rhs.denominador
                        - lhs.denominador * rhs.numerador,
                        lhs.denominador * rhs.denominador );
        return temp.simplificar();
}

Racional operator*( const Racional& lhs, const Racional& rhs )
{
        Racional temp(  lhs.numerador * rhs.numerador,
                        lhs.denominador * rhs.denominador );
        return temp.simplificar();
}

Racional operator/( const Racional& lhs, const Racional& rhs )
{
        Racional temp(  lhs.numerador * rhs.denominador,
                        lhs.denominador * rhs.numerador);
        return temp.simplificar();
}

boolean operator==( const Racional& lhs, const Racional& rhs )
{
        return (  lhs.numerador * rhs.denominador
                        == lhs.denominador * rhs.numerador );
}
```

```
inline boolean operator!=(      const Racional& lhs,
                                                const Racional& rhs )
{
        return !( lhs == rhs );
}

boolean operator>( const Racional& lhs, const Racional& rhs )
{
        return (    lhs.numerador * rhs.denominador
                        > lhs.denominador * rhs.numerador );
}

boolean operator<( const Racional& lhs, const Racional& rhs )
{
        return (    lhs.numerador * rhs.denominador
                        < lhs.denominador * rhs.numerador );
}

boolean operator<=( const Racional& lhs, const Racional& rhs )
{
        return (    lhs.numerador * rhs.denominador
                        <= lhs.denominador * rhs.numerador );
}

boolean operator>=( const Racional& lhs, const Racional& rhs )
{
        return (    lhs.numerador * rhs.denominador
                        >= lhs.denominador * rhs.numerador );
}

Racional Racional::operator-()
{
        Racional temp( -numerador, denominador );
        return temp;
}

Racional& Racional::operator+()
{
        return *this;
}

Racional& Racional::operator++()
{
        numerador += denominador;
        return *this;
}

Racional& Racional::operator--()
{
        numerador -= denominador;
        return *this;
}
```

```
Racional Racional::operator++( int )
{
        Racional temp = *this;
        numerador += denominador;
        return temp;
}

Racional Racional::operator--( int )
{
        Racional temp = *this;
        numerador -= denominador;
        return temp;
}

inline boolean Racional::operator!()
{
        return !( int( numerador ) );
}

Racional& Racional::operator+=( const Racional& rhs )
{
        numerador =     numerador * rhs.denominador
                                + denominador * rhs.numerador;
        denominador =   denominador * rhs.denominador;
        simplificar();
        return *this;
}

Racional& Racional::operator-=( const Racional& rhs )
{
        numerador =     numerador * rhs.denominador
                                - denominador * rhs.numerador;
        denominador =   denominador * rhs.denominador;
        simplificar();
        return *this;
}

Racional& Racional::operator*=( const Racional& rhs )
{
        numerador =     numerador * rhs.numerador;
        denominador =   denominador * rhs.denominador;
        simplificar();
        return *this;
}

Racional& Racional::operator/=( const Racional& rhs )
{
        numerador =     numerador * rhs.denominador;
        denominador =   denominador * rhs.numerador;
        simplificar();
        return *this;
}

float Racional::comoFloat()
```

```
{
        return (float)numerador / (float)denominador;
}

double Racional::comoDouble()
{
        return (double)numerador / (double)denominador;
}

long Racional::comoLong()
{
        return numerador / denominador;
}

int Racional::comoInt()
{
        return (int)numerador / (int)denominador;
}

Racional& Racional::simplificar()
{
        int minimo;
        int mcd = 1;        // maximo comun divisor
        if ( numerador > denominador )
                minimo = denominador;
        else
                minimo = numerador;
        for ( int i = 1; i <= minimo; i++ )
                if      ( ( numerador % i == 0 )
                        && ( denominador % i == 0 ) )
                                mcd = i;
        numerador /= mcd;
        denominador /= mcd;
        return *this;
}
```

MANEJO ELEMENTAL DE OBJETOS DE TIPO *RACIONAL*

El siguiente código es un muy elemental ejemplo de las posibilidades prácticas de uso de distintos objetos de tipo Racional, y ni siquiera agota los métodos definidos en nuestra clase.

```
// fichero:             RACIOTST.CPP
// contenido:   ejemplos de uso de la clase "Racional",
// autor:       Ricardo Devis
// derechos:    (C) INFO+ (1992)
//                      Este codigo puede copiarse libremente,
//                       con la unica restriccion de respetar
//                       el aviso de copyright.
// last mod:    03-06-93

#include "racional.cpp"
```

```
#include <iostream.h>

#define nl '\n'

int main( int, char** )
{
        Racional  resultado, dosTercios( 2, 3 ),
                          cincoMedios( 10, 4 );

        cout      << "MUESTRA EFECTO DE CONSTRUCTORES: " << nl;
        cout      << "resultado                        => "
                  << resultado << nl;
        cout      << "dosTercios( 2, 3 )               => "
                  << dosTercios << nl;
        cout      << "cincoMedios( 10 ,4 )             => "
                  << cincoMedios << nl << nl;

        cout      << "MUESTRA SOBRECARGA DE OPERADORES: " << nl;
        cout      << "dosTercios + cincoMedios         => "
                  << dosTercios + cincoMedios << nl;
        cout      << "2 + dosTercios                   => "
                  << 2 + dosTercios << nl;
        cout      << "dosTercios - 2                   => "
                  << dosTercios - 2 << nl;
        cout      << "3 / cincoMedios                  => "
                  << 3 / cincoMedios << nl;
        cout      << "cincoMedios * 3                  => "
                  << cincoMedios * 3 << nl;
        cout      << "-cincoMedios                     => "
                  << -cincoMedios << nl;
        cout      << "dosTercios - cincoMedios => "
                  << dosTercios - cincoMedios << nl;
        cout      << "dosTercios + -cincoMedios        => "
                  << dosTercios + -cincoMedios << nl;
        cout      << "+cincoMedios                     => "
                  << +cincoMedios << nl;
        cout      << "++cincoMedios                              => "
                  << ++cincoMedios << nl;
        cout      << "cincoMedios--                    => "
                  << cincoMedios-- << nl;
        cout      << "resultado = ++cincoMedios        =>resultado = ";
        cout      << (resultado = ++cincoMedios)
                  << " y cincoMedios = " << cincoMedios << nl;
        cout      << "resultado = cincoMedios--        => resultado = ";
        cout      << (resultado = cincoMedios--)
                  << " y cincoMedios = " << cincoMedios << nl;
        cout      << "resultado += cincoMedios         => "
                  << (resultado += cincoMedios) << nl;
        cout      << "resultado += 2                             => "
                  << (resultado += 2 ) << nl;
        cout      << "resultado -= dosTercios          => "
                  << (resultado -= dosTercios) << nl;
        cout      << "resultado *= cincoMedios                   => "
                  << (resultado *= cincoMedios) << nl;
```

```
        cout    << "resultado /= dosTercios             => "
                << (resultado /= dosTercios) << nl << nl;

        cout    << "MUESTRA CONVERSION A TIPOS PREDEFINIDOS: "
                << nl;
        cout    << "cincoMedios.comoInt() + 1 => "
                << (cincoMedios.comoInt() + 1) << nl;
        cout    << "cincoMedios.comoFloat() + 1.5 => "
                <<(cincoMedios.comoFloat() + 1.5) << nl;
        cout    << "cincoMedios.comoDouble() + 2.5 => "
                <<(cincoMedios.comoDouble() + 2.5) << nl;
        cout    << "cincoMedios.comoLong() + 10 => "
                << (cincoMedios.comoLong() + 10) << nl;

        return 0;
}
```

9

DE LA HERENCIA Y DERIVADOS

A estas alturas ya hemos revisado muchos de los tópicos del lenguaje C++ que nos habrán de permitir diseñar, al principio con inevitable candidez, nuestras primeras clases. De hecho, como ya vimos en la construcción de la clase *Racional*, la tradicional metodología de programación ha sufrido un vuelco (otra cuestión sería si hemos derramado algo realmente importante). ¿Quiere esto decir, pues, que ya estamos programando con técnicas de orientación-a-objetos? Bueno, un descorazonador halito de honradez me impide decir, en puridad, que sí. La verdad es que, si recordamos bien, las propiedades (o, más bien, facilidades) que caracterizan a un OOPL son: abstracción, encapsulación, herencia y polimorfismo. En la mera construcción de clases hemos involucrado solamente los conceptos de abstracción y encapsulación, así como una muy rudimentaria aproximación al polimorfismo (mediante la sobrecarga de operadores y funciones). En C++, sin embargo, los mecanismos de OOP tienen su principal apoyo en la característica denominada "herencia" y que en C++ se implementa mediante la derivación de clases: en ella se basa el mecanismo polimórfico del lenguaje, constituido por las funciones virtuales; en ella se realzan y tienen pleno sentido, también, las propiedades de abstracción y encapsulación. Apreciamos, así, que la herencia es el ojo del huracán de la OOP con respecto de C++ . No debemos olvidar, con todo, que, además de C++ y por raro que parezca, existen otros OOPL's en los que no todo tiene su exacta correspondencia: en el lenguaje SELF, por ejemplo, la herencia se sustituye por la *delegación*, una interesantísima propiedad (de posible implementación "manual" en C++), cuya explicación en detalle sobrepasa (¿cómo no?) los límites de esta introducción. Bueno, derivemos a lo práctico y heredaremos facilidad de codificación.

¿Qué es la herencia? El acto por el que un ente transfiere parte o la totalidad de sus haberes o componentes a otro u otros entes. ¿Se puede hablar de herencia en sistemas de objetos? ¡Naturalmente! De la misma -y a veces desafortunada- forma que los hijos poseen una parte genética de sus padres, los objetos a veces poseen una parte "heredada" (y lo cierto es que esta visualización se corresponde con la realidad física de la herencia simple). ¿Un objeto puede, pues, heredar propiedades de otro u otros? ¡No! Como ya se ha recordado repetidas veces, C++ es un lenguaje cuya esencia dual está representada por las clases y los objetos, limitándose estos últimos a ser meras instanciaciones de aquéllas. O sea, propiedades como abstracción, encapsulación, etc., se aplican a las clases, y no a los objetos. La herencia es, por tanto, una propiedad únicamente circunscrita a las clases. Naturalmente los objetos, instanciaciones de las clases inmersas en el sistema de derivación, se beneficiarán de esta circunstancia, pero el lector debe tener muy claro que lo que se hereda no son los valores específicos de los miembros de uno o más objetos, sino las "estructuras vacías" que conforman las clases a que pertenecen. ¿Como se implementa la herencia en C++? Mediante el mecanismo conocido como *derivación de clases*: a través de la aplicación de la sintaxis que seguidamente veremos, en la definición de una clase se puede "enlazar" ésta a otra ya definida, consiguiendo que aquélla se constituya en lo que se denomina *clase base*, mientras que ésta pasa a ser una clase *derivada*, heredando, junto con ciertos protocolos de acceso, distintos miembros (datos y funciones) de la clase base. ¿Cualquier clase se puede "hacer derivar" de otra dada? Pues sí: con la exclusión de la autoderivación, en el momento de definir una clase se puede expresamente indicar la voluntad de desear heredar de una o más clases, con la única precaución de que estén ya definidas. ¿Una o más? ¿Entonces una clase derivada puede depender de más de una clase base? ¡Efectivamente! La herencia puede ser *simple* (derivación de una sola clase base) o *múltiple* (derivación de distintas clases base). C++ nació[21], de hecho, solamente con la herencia simple, incorporándose la múltiple a partir de AT&T 2.0. En lo que sigue hablaremos, básicamente, de la herencia simple, aunque también veremos algunos rudimentos de uso de la herencia múltiple (como también repasaremos muchos de sus problemas). Bueno, pero ¿para qué demonios sirve la herencia? Muy sencillo: para permitir la tan cacareada *reutilización* del código. En efecto: la porción física, correspondiente a cada una de las clases base, que se añade a la clase derivada es el resultado de la reutilización -sin tener que reescribirlo- de parte del código usado en cada una de las clases base. ¿Es entonces la derivación de clases únicamente una instrumentación de C++ para procurar la reutilización del código? Bien, la verdad es que estamos considerando la cuestión desde un ángulo erróneo: el mecanismo de derivación da lugar a jerarquías más o menos complejas de

[21] Volvemos aquí al gastado tema de la prevalencia al considerar los posibles estándares del lenguaje. C++ nació, de hecho, como una implementación particular de AT&T, independientemente de que ARM sirviera después como documento base para el comité ANSI C++. No cansaré, pues, más al lector.

clases, procurando relaciones de distinto tipo y cualificación entre las mismas y posibilitando, así, la asunción, para el subsistema así formado, de los distintos comportamientos que conforman lo que se conoce como OOP. Naturalmente que se reutiliza código, aunque es posible pensar en un particular esquema de derivación en el que no se reutilice una sóla línea ya escrita. La derivación posibilita, además, la implementación completa de la cualidad de polimorfismo (que parcialmente vimos aplicada en la sobrecarga de funciones) mediante las funciones virtuales. ¿Existen distintos tipos de derivación de clases? Ciertamente; como distintas formas hay de testar. El tipo de derivación depende, únicamente, del nivel de acceso -o de restricción del acceso- elegido para la misma. Existen, pues, tres tipos de derivación: pública (*public*), protegida (*protected*) y privada (*private*). Esto recordará al lector la cualificación de acceso de las secciones de una clase (siendo parejo, en el fondo, el funcionamiento en ambos estadios), que siempre hemos visto únicamente como *public* y *private*, pero que ahora podemos ampliar formalmente también a *protected*. La sección protegida de una clase, a los solos efectos de su consideración como clase aislada no participante en un esquema de derivación, se considerará como privada: esto es, las instanciaciones de tal clase no tendrán acceso a lo en ese bloque contenido. Esto cambia cuando de tal clase se derivan otras, pero ya lo veremos con más detalle en su momento.

Bien, contestadas las preguntas básicas comenzaremos la revisión de la herencia en C++: primero la sintaxis; luego el uso; más tarde el abuso; seguirá la racionalidad; y, para terminar, el ejemplo.

HERENCIA SIMPLE: UN LOBO CON PIEL DE MELOCOTÓN

Como ya sabemos, si en la definición de una clase no se indica etiqueta alguna de acceso, la cualificación por defecto es *private* (como en los *struct*s es *public*), reforzando así la tendencia natural en C++ hacia lo que se denomina ocultación de la información (*information hiding*). De la misma manera, si en una derivación no se indica etiqueta de acceso, tal derivación se entenderá *private*, reforzando de esta manera, también, la tendencia en C++ a usar tal tipo de propagación derivativa[22]. De cualquier forma, para evitar innecesarias confusiones, siempre codificaré las etiquetas de acceso en las derivaciones. Veamos, sin más dilación, la sintaxis de derivación:

[22] ¡Oops! De hecho la afirmación contraria se acerca muchísimo más a la realidad práctica: los mecanismos de ligazón dinámica y orientación-a-objetos de C++ se asimilan sobremanera con la derivación pública. En la sección 11.2 de ARM se afirma, refiriéndose a la derivación de clases: "Probablemente fue un error definir un especificador de acceso por defecto.". Bien, el lector lo tiene claro: para evitar problemas en el presente y en el futuro, codifique siempre expresamente la cualificación de acceso en derivación.

```
class Base {
public:             // siguen miembros públicos
protected:                  // siguen miembros protegidos
private:            // siguen miembros privados
};                          // clase Base ya definida

class Derivada : public Base { // derivación pública de Base
public:     // siguen miembros públicos de clase Derivada
protected:          // siguen miembros protegidos de clase Derivada
private:    // siguen miembros privados de clase Derivada
};
```

Notamos que simplemente se añaden, entre el nombre de la clase y la llave de apertura de la definición de la misma, dos puntos y el nombre de la clase base antecedido por el cualificador de acceso deseado para la derivación. El resto de la clase Derivada es, en principio, igual al de una clase "normal". Esta simple adición va a sumergirnos, sin embargo, en el análisis de distintas consideraciones hasta ahora sin sentido. Pensemos, por ejemplo, que, de alguna manera, se está añadiendo una porción de la clase Base a la clase Derivada. Así, si definimos un constructor para la clase Derivada, ¿qué tendremos que hacer para construir también la porción de la clase Base? ¿Cual será, lógicamente, por otra parte, el funcionamiento con respecto al destructor? ¿Cuándo, en definitiva, deberá ser usado uno u otro mecanismo de derivación? ¿Cómo escoger las clases base apropiadas? Etc., etc., etc. Bien, antes de ver el qué, el cuando o el por qué, le echaremos un vistazo al cómo. Y dado que la sintaxis es manifiestamente sencilla, éste se limitará a la explicación de las tres clases posibles de derivación.

DERIVACIÓN PÚBLICA: LA FACILIDAD DE LA HERENCIA

No olvidemos que lo que se consigue con la derivación es, en esencia, añadir una parte de una clase a otra. Tendría, así, poco o ningún sentido la aplicación de una clase base sólo a una clase derivada, pues en este caso quizá lo más práctico hubiera sido añadir simplemente la implementación de la clase base como código a la clase derivada. Se supone, pues, que una clase base lo será de varias derivadas (o, al menos, tendrá posibilidad efectiva de serlo). O sea, de alguna manera en las clases base encapsularíamos las características comunes susceptibles de ser usadas por las clases derivadas[23]. Pero, bueno, veámoslo en un ejemplo (suponiendo el planeta habitado por un muy lineal tipo de animal):

[23] De hecho, en la programación efectiva con C++, lo más frecuente es que, habiendo reunido más de una clase con características comunes, se haga abstracción de éstas y se cree una superclase o clase base de las mismas y que se insertaría con naturalidad en la posible jerarquía de clases ya existente. Naturalmente un diseño adecuado identificaría apropiadamente las relaciones iniciales de derivación.

```
class Persona {
public:
        char* nombre() const {
                return nombre_;
        }
protected:
        Persona() {
                nombre_ = new char[ 1 ];
                nombre_ = '\0';
                cout    << "Constructor por defecto de Persona"
                        << '\n';
        }
        Persona( char* unNombre ) {
                nombre_ = new char[ strlen( unNombre ) + 1 ];
                strcpy( nombre_, unNombre );
                cout    << "Constructor con un argumento de Persona"
                        << '\n';
        }
        ~Persona() {
                delete [] nombre_;
                cout << "Destructor de Persona" << '\n';
        }
private:
        char* nombre_;
};

class Profesional : public Persona {
public:
        void imprimeDatos() const {
                cout    << "Nombre: \t" << nombre() << '\n';
                cout    << "Años ejercicio: \t"
                        << anosEjercicio << '\n';
        }
private:
        int anosEjercicio;
};
```

¿Qué novedades podemos apreciar? ¿Y cuáles no podemos? Creo que merece la pena dedicar algunas líneas a estas cuitas, aunque algunas no tengan directamente que ver con la herencia. Bueno, examinemos en primer lugar la clase *Persona*. Hemos declarado dos constructores como protegidos: esto quiere decir que serán considerados *private* para la clase *Persona*. De esta manera el siguiente código

```
Persona miJefe( "María Cristina" );        // error
```

sería calificado como error: al no poder acceder un objeto a los miembros privados de su clase, ¡no podemos construir objetos de tipo *Persona*! Pero no hay que ser dramáticos: no podemos construir los objetos "desde el exterior", pero sí dentro del ámbito de las clases derivadas. ¿Y cómo ...? Enseguida vamos a ello. Recabemos antes en el detalle de los constructores: en el cuerpo del constructor con un argumento, un puntero a *char*, primero

reservamos espacio de la memoria de almacenamiento libre (la longitud de la cadena del argumento más el espacio del bit nulo final) y asignamos su dirección al puntero *nombre_*, para después copiar nuestra cadena en tal espacio. ¿Y no sería más facil -podría pensar algún inquieto lector- simplemente asignar la dirección de la cadena del argumento al miembro *nombre_*? No sería más fácil: sólo más corto y mucho más peligroso. Imaginemos la siguiente situación:

```
class Persona {
protected:
        Persona( char* unNombre ) {
                nombre_ = unNombre;          // peligro!!
        }
        // …
};
char* maestro = "Ferrer Guardia";
void Profesional::algunaFuncion()
{
        Persona miIdolo( maestro );
        // …
        cout << miIdolo.nombre(); // imprime 'Ferrer Guardia'
        // con la siguiente línea vulneramos
        //  nuestro cuidado control de acceso
        *maestro = "Friedrich Nietzsche";
        cout << miIdolo.nombre(); // imprime 'Friedrich Nietzsche'
}
```

Realmente esto no funciona, así que descubrimos como buena la primera codificación. Nótese también, incidentalmente, que no se ha señalado como errónea la aplicación del constructor de la clase *Persona* en el ámbito de la clase *Profesional*, tal y como ya se había anunciado y se explicará después. En cuanto al constructor por defecto, ¿por qué no se ha codificado con el cuerpo vacío, lo que parece suficientemente apropiado? Por algo que se conoce como *coherencia* o *completitud* del interfaz: como esperamos que todas las *Personas* posean un nombre, usaremos la función *nombre()* sin especiales cuidados, esperando que devuelva la información adecuada. ¿Qué ocurriría, sin embargo, si el cuerpo de nuestro constructor por defecto está vacío? Pues que el resultado de la llamada a la función *nombre()* de un objeto creado mediante el constructor por defecto sería indefinido. Codificando como lo hemos hecho garantizamos la seguridad de uso de tal función. Naturalmente estamos obviando aquí, como se indicó en capítulos anteriores, que al usar de la memoria de almacenamiento libre, deberíamos proveer a la clase de sus propias versiones del constructor de copia y del operador de asignación, pero esta es una cuestión que relegaremos a la construcción de nuestra clase *String*. Haremos, pues, abstracción de estas carencias a efectos de comentar con más simplicidad el fenómeno de la herencia. Una última cosa: el lector se preguntará el sentido de las sentencias de impresión en los constructores y destructor. Bueno, esto nos va a ayudar a identificar cuándo entra en acción cada uno de ellos, pues

existen ciertos hilos conductores a veces en absoluto evidentes para el lector no experimentado.

Examinemos ahora, levemente, la clase *Profesional*. Habremos de notar, al menos, dos cosas: la clase carece de constructor (explícito), y en su única función miembro se hace referencia a lo que parecen dos miembros, *anosEjercicio* (observen aquí la cruel y, por otro lado, lógica prevalencia en informática de una lengua foránea) y *nombre()*. Pero, ¡*nombre()* no aparece en la definición de la clase *Profesional*! ¡Ciertamente! Es una función miembro heredada de la clase *Persona*, a la que sí efectivamente pertenece. Vamos a ver, por fin, cómo funciona el mecanismo derivativo.

Aunque la imagen gráfica de la derivación de clases (la adición de la porción de código de la clase base a la clase derivada) es acertada, quizá sea también práctico, a efectos de ámbitos y prevalencias, pensar en la derivación como en una autorización para que, a través del filtro de las clases sucesivamente derivadas, se pueda acceder a parte de la clase base. Así, aunque se traspase a la clase derivada una porción correspondiente a la totalidad de la clase base, la parte *private* de la clase base no será accesible en ningún caso. Por la derivación únicamente se traspasa, pues, la posibilidad de acceso a las secciones no privadas de la clase base: o sea, las públicas y las protegidas. ¿Cuál es, entonces, el nivel de acceso a estas secciones adicionales? Depende del tipo de derivación. Pensemos en el cualificador de derivación como en un cristal que tamiza el acceso, desde la transparencia a la negritud, desde las clases derivadas y sus objetos a las secciones de la clase base. La derivación pública es el cristal transparente que mantiene el nivel de acceso ya establecido para los miembros no privados de la clase base. En nuestro caso podemos formarnos la siguiente representación intuitiva: a nuestra clase *Profesional* simplemente se le han añadido las secciones *protected* y *public* (naturalmente también se añade físicamente la sección *private*, aunque en ningún caso se permite el acceso a la misma) de la clase *Persona*[24]. De esta manera, como ya sabemos, desde el protocolo de descripción de la clase *Profesional* podremos acceder a los miembros de ambas secciones de la clase base; desde los objetos de tipo *Profesional* podremos acceder únicamente a la sección pública de la clase base (obviamos aquí, naturalmente, el acceso a las secciones de la propia clase derivada).

Siguiendo con el esquema lúdico propuesto, el lector podría imaginarse la derivación *protected* como un cristal gris, del mismo gris que tendrían las secciones protegidas de las clases si a éstas se las dotara de color, que tamizará las secciones *public* y *protected* de la clase base y convertirá el

[24] En realidad esto es una mera regla mnemotécnica que el lector no debe tomar al pie de la letra, pues aparte del nivel de acceso habría que considerar las cuestiones de prevalencia y solapamiento de miembros (pensemos, por ejemplo, en un miembro de una clase base y otro de una clase derivada de ésta que compartan el mismo identificador: ¡problema!).

acceso para ambas a través de la clase derivada en *¡protected!*. El lector ya podrá adivinar que la derivación *private*, cual cristal oscuro, originará que, a efectos del acceso, la clase derivada considerará como *private* las secciones *public* y *protected* de la clase base. Pero esto lo veremos un poco más adelante: volvamos a la derivación pública.

Según lo expuesto, ahora podemos comprender que desde el interior de la clase *Profesional* tenemos acceso a la función miembro *nombre()*, como si ésta, a los solos efectos del nivel de acceso, hubiera sido directamente declarada como pública en la clase derivada. Hubiera sido un error, sin embargo, codificar lo siguiente:

```
class Profesional : public Persona {
public:
        void imprimeDatos() const {
                cout    << "Nombre: \t" << nombre_ << '\n';   // error
                cout    << "Años ejercicio: \t"
                        << anosEjercicio << '\n';
        }
        // sigue definición de clase
};
```

pues el dato miembro *nombre_* no pertenece a una sección con acceso *private* desde la clase *Profesional*, sino que pertenece a la sección *private* de la clase base (¡que no es lo mismo!), y esta sección es, en cualquier caso y tipo de derivación, inaccesible desde las clases derivadas (la porción está ahí, físicamente, pero no puede ser accedida si no es a través de las secciones pública y protegida de su propia clase).

CONSTRUCTORES DE CLASES EN JERARQUÍAS DE DERIVACIÓN

Bien, traspasemos ahora nuestra atención al siempre espinoso tema de los constructores. Notamos que la clase *Profesional* carece de constructor explícito, por lo que, como sabemos, el sistema la proveerá con uno implícito. De acuerdo: pero, ¿cómo se construye la porción de la clase base?. Examinemos el siguiente simplísimo ejemplo:

```
int main( int, char** )
{
        Profesional sexadorDePollos;
        return 0;
}
```

¿Puede el lector adivinar qué ocurre cuando se ejecuta este código? Veamos la salida:

```
Constructor por defecto de Persona
Destructor de Persona
```

¿Por qué? La información para construir una parte de nuestro objeto reside en la clase base *Persona*, por lo que antes de aplicar el constructor implícito por defecto para la clase derivada, el sistema exige que se construya la porción de la clase base, y para esto busca en ésta un constructor por defecto (que en nuestro caso sí existe) y lo ejecuta, para aplicar después la construcción de la porción correspondiente a la clase derivada. Recursivamente, si nuestra clase base fuera, a su vez, derivada de otra, antes de usar el constructor de ésta se forzaría la construcción de la porción de su clase base, etc. Pero quizá estamos en un estadio demasiado elemental para entrar en detalles. Dotemos, pues, a nuestra clase *Profesional* de constructores y destructor expresos:

```
class Profesional : public Persona {
public:
        Profesional();
        Profesional( int );
        ~Profesional() {
                cout << "Destructor de Profesional" << '\n';
        }
        // sigue resto definición de clase
};

Profesional::Profesional() : anosEjercicio( 0 )
{
        cout << "Constructor por defecto de Profesional" << '\n';
}

Profesional::Profesional( int experiencia )
        : anosEjercicio( experiencia )
{
        cout    << "Constructor con un argumento de Profesional"
                << '\n';
}
```

Si con las adiciones de código expuestas ejecutamos de nuevo el anterior programita, la salida cambiará de esta forma:

```
Constructor por defecto de Persona
Constructor por defecto de Profesional
Destructor de Profesional
Destructor de Persona
```

Ahora podemos apreciar mejor una cuestión de orden: primero se construye la porción del objeto correspondiente a sus clases base (lo cual es lógico, pues pensemos que en el constructor de la clase derivada se puede hacer referencia a miembros de sus clases base, y si estos aún no han sido construidos entonces: ¡problema!); los destructores, sin embargo, siguen un orden inverso: primero se destruye la porción del objeto correspondiente a la clase más derivada y después, sucesivamente, las porciones de las clases base en jerarquía. Si pensamos un poco en este orden inamovible, apercibiremos una fácil consecuencia: en el cuerpo de un constructor de una

clase base no se puede hacer referencia a un miembro (y, por supuesto, a un constructor) de una clase derivada, pues ésta, cuando se está ejecutando el cuerpo de tal constructor de la clase base, ¡todavía no ha sido construida!.

INICIALIZACIÓN DE CLASES BASE

De acuerdo, de acuerdo: las nubes desaparecen, poco a poco, del horizonte de la derivación, pero lo cierto es que todavía no hemos solucionado un problema: en cualquier caso construimos una porción de clase base en nuestros objetos, pero ésta contiene, siempre, un valor por defecto. En realidad nosotros deberíamos querer construir un *Profesional* como una *Persona*, con su propio nombre, con determinados años de experiencia. Pero, ¿cómo podemos inicializar el miembro *nombre_* , si desde la clase derivada no tenemos acceso al mismo? Fácilmente: ¿cómo se accede a los miembros "escondidos" de una clase? ¡A través de las funciones miembro que conforman el interfaz de la clase! ¿Y a través de qué función miembro podríamos, en nuestro caso, acceder a *nombre_*? Pues a través de un constructor de la clase base, naturalmente, que es la única función miembro con la que contamos. ¿Y no sería mejor dotar a la clase base de una función de acceso *public* que permitiera la modificación del dato miembro *nombre_* , como por ejemplo

```
Persona::modificaNombre( char* unNombre )
{
        delete nombre_;
        nombre_ = new char[ strlen( unNombre ) + 1 ];
        strcpy( nombre_, unNombre );
        cout << "Función miembro modificaNombre" << '\n';
}
```

? Pues realmente no: montemos una pequeña simulación, añadiendo un nuevo constructor a nuestra clase derivada:

```
class Profesional : public Persona {
public:
        Profesional( char*, int );
        // sigue resto definición de clase
};

Profesional::Profesional( char* unNombre, int experiencia )
        : anosEjercicio( experiencia )
{
        modificaNombre( unNombre );
        cout    << "Constructor con dos argumentos de Persona"
                << '\n';
}
```

¿Solucionado todo? ¡Más bien no! Veamos la salida que arroja, con esta nueva adición, la siguiente variación de nuestro simpático programita:

```
int main( int, char** )
```

```
{
        Profesional sexadorDePollos( "Mario", 20 );
        return 0;
}
```

He aquí el resultado:

```
Constructor por defecto de Persona
Constructor con dos argumentos de Profesional
Función miembro modificaNombre
Destructor de Profesional
Destructor de Persona
```

Vemos que nuestra implementación no es muy eficiente: la porción de clase base se sigue construyendo con el constructor por defecto de *Persona*, y más tarde se usa una función miembro para cambiar el nombre. ¿No podríamos trocar estos dos procesos en uno solo? Por supuesto, usando, como ya se apuntó, el constructor con un argumento de la clase base. Pero, ¿cómo? ¿cómo podemos aplicar tal constructor exactamente al objeto derivado que estamos construyendo? Pues, siguiendo un razonamiento similar al observado respecto de la inicialización de miembros en constructores, mediante su inclusión expresa en la lista de inicialización del constructor de la clase derivada. Desechemos, pues, la función *modificaNombre(...)* e intentemos lo apuntado:

```
Profesional::Profesional( char* unNombre, int experiencia )
        : Persona( unNombre ), anosEjercicio( experiencia )
{
        cout    << "Constructor con dos argumentos de Profesional"
                << '\n';
}
```

La salida del último programa quedaría ahora de la siguiente guisa:

```
Constructor con un argumento de Persona
Constructor con dos argumentos de Profesional
Destructor de Profesional
Destructor de Persona
```

Con esta sintaxis lo que hacemos es indicar de forma expresa al constructor de nuestro objeto qué constructor en concreto de la clase base debe utilizar. Naturalmente sólo podemos hacer esto con respecto a la clase base de la nuestra, pero sin poder irnos más lejos en la escala derivativa. Intentemos ahora, para aclarar definitivamente las cuestiones de orden, una derivación adicional:

```
class Directivo : public Profesional {
public:
        Directivo( long unSalario = 0) : salario( unSalario ) {
                cout    << "Constructor por defecto de Directivo"
                        << '\n';
```

```
        }
        Directivo(      char* unNombre, int experiencia,
                        long unSalario)
                                : Profesional( unNombre, experiencia ),
                                  salario( unSalario )          {
        cout    << "Constructor con tres argumentos de Directivo"
                << '\n';
        }
        ~Directivo() {
                cout << "Destructor de Directivo" << '\n';
        }
private:
        long salario;
};
```

Como vemos, en la lista de inicialización del constructor con tres argumentos de *Directivo* aparece el constructor de su clase base y la inicialización de su único dato miembro. Si hubiéramos intentando forzar el uso de un determinado constructor de la clase *Persona*, el compilador lo señalaría como error indicando que *Persona* no es una clase base de *Directivo*. Veamos qué ocurre ahora con nuestro programilla, convenientemente modificado:

```
int main( int, char** )
{
        Profesional gigolo( "Maurice", 7 );
        Directivo directorComercial( "Ramón", 20, 50000 );
        return 0;
}
```

He aquí el resultado (los comentarios son míos, a efectos de clarificación):

```
// construcción del objeto Profesional
Constructor con un argumento de Persona
Constructor con dos argumentos de Profesional
// construcción del objeto Directivo
Constructor con un argumento de Persona
Constructor con dos argumentos de Profesional
Constructor con tres argumentos de Directivo
// seguidamente se destruyen ambos objetos,
//  pero el orden de destrucción no tiene
//  por qué ser el que aquí se va a indicar
// por ejemplo: primero se puede destruir el objeto Directivo
// tal y como ocurriría con el compilador Borland C++
Destructor de Directivo
Destructor de Profesional
Destructor de Persona
// ahora se destruiría (en nuestro supuesto)
//  el objeto Profesional
Destructor de Profesional
Destructor de Persona
```

El esquema de orden de intervención de constructores y destructores, aun ampliado con una nueva clase, sigue siendo el mismo. Cabe comentar aquí, pues viene a colación, lo apuntado en las líneas anteriores sobre el orden en que se destruirán los objetos (que no tiene que ver con el orden en que se aplicarán los destructores en una clase derivada de otras, como ya sabemos). Bien, ¿C++ nos depara otra sorpresa? Lo cierto es que esto no tiene que ver directamente con la herencia, pero no está de más anotarlo: ¿cuando entra en acción el destructor de un objeto? En cualquier momento desde el último uso del objeto en el código hasta el final del ámbito del mismo: no hay momento prefijado ni prevalencia de objetos, por lo que el lenguaje no asegura que, en el caso anterior, el objeto *Directivo* deba destruirse antes que el objeto *Persona*. Realmente esta cuestión depende únicamente del compilador.

DERIVACIONES NO-PÚBLICAS: SÓLO PARA TUS OJOS

Como ya hemos apuntado, la derivación pública ocasiona que las secciones no-privadas de la clase base puedan ser abordadas con su mismo nivel de acceso, pero con respecto a la clase derivada. O sea, las secciones *public* y *protected* de la clase base tendrán cualificación de acceso *public* y *protected*, recpectivamente, desde la clase derivada (recordemos el símil del cristal transparente, que mantiene invariantes los niveles de acceso).

La derivación puede ser, también, **protected** (*lisez* cristal gris), ocasionando que las secciones no-privadas de la clase base se conviertan en secciones *protegidas* con respecto de la clase derivada. O sea, serán privadas para la clase derivada, aunque podrán ser accedidas, al menos, por las clases (no por los objetos) derivadas de ésta. Así, los miembros de las secciones *public* y *protected* de la clase base se convierten en miembros *protected* de la clase derivada.

La derivación podría también cualificarse, al fin, como **private** (*lisez* cristal negro), siendo así que las secciones no-privadas de la clase base se convertirán en secciones *private* con respecto de la clase derivada. Esto es, serán privadas para la clase derivada, e inaccesibles para las clases derivadas de ésta. De esta forma, los miembros de las secciones *public* y *protected* de la clase base se convertirán en miembros *private* de la clase derivada.

Queda claro, pues, que la derivación pública es la única que permite, en todo caso, a los objetos de clases derivadas acceder a determinados miembros de sus clases base (los insertados en la sección *public* de éstas). Intuitivamente podemos apreciar, también, que las derivaciones no-públicas sirven, sobre todo, para la reutilización "interna" del código ya escrito en clases bases.

Bien, vayamos a otra cuestión. Aun ceñiéndonos sólo a la herencia simple, imaginemos el siguiente panorama: de una clase A se deriva públicamente

otra B, y de ésta se deriva otra, C, con cualificación protegida; y de ésta, a su vez, se deriva privadamente otra D; y de ésta, de nuevo, se deriva como protegida una nueva clase E, y así *ad nauseam*. ¿Y si ahora preguntamos: a qué accede D, a qué no puede acceder E, a qué ...? Despacio, despacio. Lo cierto es que si nos encontramos en la realidad con una algarabía semejante, lo primero que debemos hacer es inquirir de qué sanatorio mental escapó el autor del desaguisado, para devolverlo allí con rapidez, por la fuerza si es necesario. ¿Quiere decir esto que no podemos mezclar distintos tipos de derivación en una jerarquía de clases? ¡Ni mucho menos! Son frecuentes derivaciones del tipo público-público-público-privado ó privado-público--público-público, pero, como es fácilmente apreciable, estos esquemas ofrecen, en comparación, una gran limpieza. Volvemos, pues, a una muy repetida apreciación sobre C++: el lenguaje proporciona muchos instrumentos que refuerzan su gran flexibilidad, pero a la vez insta a los usuarios a utilizarlos con mucha prudencia. Por otro lado es momento de anunciar algo que al lector le puede parecer, cuando menos, sorprendente: si se usa la derivación de clases como un mero mecanismo formal, más o menos ingenioso, de ahorrar trabajo al desarrollador, se estará incurriendo en el peor error de todos, como si usáramos un martillo pilón para partir la cáscara de un huevo, con la desventaja adicional del coste eléctrico que supone cada golpe. Aún más: la siguiente consecuencia de esta actitud será, probablemente, que el mismo desarrollador abandone el uso de los mecanismos de derivación en su código, cansado de sus pocas ventajas y muchas dificultades. ¿Se sorprenden? Pues sepan que, en la actualidad, la mayoría de los usuarios de C++ lo utilizan como un "mejor C": esto quiere decir que sólo una pequeña parte de los mismos utilizan las clases, y de éstos únicamente una pequeñísima porción usa de los mecanismos de derivación (y esto no son imaginaciones mías, sino que se ha puesto de manifiesto en los últimos congresos, seminarios y jornadas sobre C++ celebrados en USA)[25]. Teniendo en cuenta que la OOP en C++ se sostiene, sobre todo, por la derivación de clases y las funciones virtuales (ligadas a aquélla), resulta así que "mucho C++ y poco OOP". Entonces, ¿es necesario meditar durante varios años en un almacén de cereales para poder usar con juicio los mecanismos de derivación en C++? No, no me han entendido. No es que sea particularmente difícil comprender tales mecanismos. Lo verdaderamente difícil es comprender el tipo de relaciones del mundo real que pueden ser capturados con los mismos. Si uno llega a comprender el sustrato semántico que anima las distintas formas de derivar clases, a la vez que se compromete con el buen diseño de las clases individuales y de sus interfaces, buena parte del trabajo ya estará hecho.

[25] Dada la apreciable segmentación en el uso del lenguaje, se han llegado a establecer tres diferentes niveles de uso de C++: "mejor C" o "C con chequeo de tipos", que utiliza todas las características de C++ no relacionadas con las clases; "C++ basado en Objetos", que usa de las clases, aunque no de la derivación; y, por último, "C++ completo", que abarca la totalidad del lenguaje.

Volviendo a lo anterior (a las jerarquías con derivaciones mixtas), el lector deberá observar la siguiente norma secuencial: primero aplicará la primera derivación y retendrá el nuevo nivel de acceso de las secciones; sobre estos nuevos niveles de acceso aplicará la segunda derivación, y así sucesivamente. En la práctica no es difícil, y sólo requiere cuidado.

Seguidamente podremos ver, no sé si para bien o para mal, un esquema de código en el que se explicitan diversos niveles de acceso en derivación, aunque sin agotar todas las posibilidades (que necesitarían de un enorme número de páginas y que resultarían de todavía más dudosa utilidad para el sufrido lector):

```
class Base {
public:
        void pruebaAcceso();
        void pubBf() {}
protected:
        void proBf() {}
private:
        void priBf() {}
};

class PublicDerived : public Base {
        friend class FriendPublicDerived;
public:
        void pruebaAcceso();
        void pubPUBDf() {}
protected:
        void proPUBDf() {}
private:
        void priPUBDf() {}
};

class ProtectedDerived : protected Base {
        friend class FriendProtectedDerived;
public:
        void pruebaAcceso();
        void pubPRODf() {}
protected:
        void proPRODf() {}
private:
        void priPRODf() {}
};

class PrivateDerived : private Base {
        friend class FriendPrivateDerived;
public:
        void pruebaAcceso();
        void pubPRIDf() {}
protected:
        void proPRIDf() {}
private:
        void priPRIDf() {}
```

```
};

class FriendPublicDerived {
public:
        void pruebaAcceso();
        PublicDerived objetoPublicDerived;
};

class FriendProtectedDerived {
public:
        void pruebaAcceso();
        ProtectedDerived objetoProtectedDerived;
};

class FriendPrivateDerived {
public:
        void pruebaAcceso();
        PrivateDerived objetoPrivateDerived;
};

void Base::pruebaAcceso()
{
        pubBf();
        proBf();
        priBf();
}

void PublicDerived::pruebaAcceso()
{
        pubBf();
        proBf();
        priBf();            // error
}

void ProtectedDerived::pruebaAcceso()
{
        pubBf();
        proBf();
        priBf();            // error
}

void PrivateDerived::pruebaAcceso()
{
        pubBf();
        proBf();
        priBf();            // error
}

void FriendPublicDerived::pruebaAcceso()
{
        PublicDerived objetoPublicDerived;
        objetoPublicDerived.pubBf();
        objetoPublicDerived.proBf();
        objetoPublicDerived.priBf();        // error
```

```
        objetoPublicDerived.pubPUBDf();
        objetoPublicDerived.proPUBDf();
        objetoPublicDerived.priPUBDf();
}

void FriendProtectedDerived::pruebaAcceso()
{
        ProtectedDerived objetoProtectedDerived;
        objetoProtectedDerived.pubBf();
        objetoProtectedDerived.proBf();
        objetoProtectedDerived.priBf();          // error
        objetoProtectedDerived.pubPRODf();
        objetoProtectedDerived.proPRODf();
        objetoProtectedDerived.priPRODf();
}

void FriendPrivateDerived::pruebaAcceso()
{
        PrivateDerived objetoPrivateDerived;
        objetoPrivateDerived.pubBf();
        objetoPrivateDerived.proBf();
        objetoPrivateDerived.priBf();            // error
        objetoPrivateDerived.pubPRIDf();
        objetoPrivateDerived.proPRIDf();
        objetoPrivateDerived.priPRIDf();
}


int main( int, char** )
{
        Base objetoBase;
        objetoBase.pubBf();
        objetoBase.proBf();                      // error
        objetoBase.priBf();              // error

        PublicDerived objetoPublicDerived;
        objetoPublicDerived.pubBf();
        objetoPublicDerived.proBf();             // error
        objetoPublicDerived.priBf();             // error

        ProtectedDerived objetoProtectedDerived;
        objetoProtectedDerived.pubBf();                  // error
        objetoProtectedDerived.proBf();                  // error
        objetoProtectedDerived.priBf();                  // error

        PrivateDerived objetoPrivateDerived;
        objetoPrivateDerived.pubBf();                    // error
        objetoPrivateDerived.proBf();                    // error
        objetoPrivateDerived.priBf();            // error

        FriendPublicDerived objetoFriendPublicDerived;
        objetoFriendPublicDerived.
                objetoPublicDerived.pubBf();
        objetoFriendPublicDerived.
```

```
        objetoPublicDerived.proBf();            // error
    objetoFriendPublicDerived.
        objetoPublicDerived.priBf();            // error
    objetoFriendPublicDerived.
        objetoPublicDerived.pubPUBDf();
    objetoFriendPublicDerived.
        objetoPublicDerived.proPUBDf();         // error
    objetoFriendPublicDerived.
        objetoPublicDerived.priPUBDf();         // error

    FriendProtectedDerived objetoFriendProtectedDerived;
    objetoFriendProtectedDerived.
        objetoProtectedDerived.pubBf();         // error
    objetoFriendProtectedDerived.
        objetoProtectedDerived.proBf();         // error
    objetoFriendProtectedDerived.
        objetoProtectedDerived.priBf();         // error
    objetoFriendProtectedDerived.
        objetoProtectedDerived.pubPRODf();
    objetoFriendProtectedDerived.
        objetoProtectedDerived.proPRODf();      // error
    objetoFriendProtectedDerived.
        objetoProtectedDerived.priPRODf();      // error

    FriendPrivateDerived objetoFriendPrivateDerived;
    objetoFriendPrivateDerived.
        objetoPrivateDerived.pubBf();                   // error
    objetoFriendPrivateDerived.
        objetoPrivateDerived.proBf();                   // error
    objetoFriendPrivateDerived.
        objetoPrivateDerived.priBf();           // error
    objetoFriendPrivateDerived.
        objetoPrivateDerived.pubPRIDf();
    objetoFriendPrivateDerived.
        objetoPrivateDerived.proPRIDf();        // error
    objetoFriendPrivateDerived.
        objetoPrivateDerived.priPRIDf();        // error

    return 0;
}
```

CONVERSIONES ESTÁNDAR BAJO DERIVACIÓN

Bien, volvamos a la representación física de la herencia y apliquémosla a un objeto del tipo *Directivo*: tal objeto estaría compuesto de tres partes, correspondientes a las tres clases que con el mismo tienen que ver: la clase *Directivo*, a que directamente pertenece, y las clases *Profesional*, de la que aquélla deriva, y *Persona*, de la que *Profesional* deriva a su vez. Tan exacta es esta partición física del objeto en porciones que realmente se puede acceder al mismo … ¡a través de tres punteros distintos! Así, por ejemplo, se cumpliría que:

```
Directivo* punteroADirectivo = new Directivo(     "Emilio",
                                                                    2, 40000 );
Profesional*        punteroAPorcionProfesionalDeDirectivo
                            = (Profesional*) punteroADirectivo;
Persona*                    punteroAPorcionPersonaDeDirectivo
                            = (Persona*) punteroADirectivo;

punteroAProcionPersonaDeDirectivo->imprimeDatos();      // OK
punteroADirectivo->imprimeDatos();                              // OK
punteroAPorcionProfesionalDeDirectivo->imprimeDatos(); // OK
```

De hecho, esta codificación viene a expresar algo así como: "consideremos el objeto de tipo *Directivo* desde distintos puntos de vista". ¡Demonios! ¿Quiere esto decir que podemos esperar resultados diferentes dependiendo de la parte de un objeto a través de la que accedemos al mismo? ¡Efectivamente! Aunque sería mejor exponerlo de la siguiente forma: tenemos la posibilidad física de cambiar el comportamiento de un objeto dependiendo de la cualificación del puntero o referencia desde el que lo accedamos, pero tenemos también la obligación (dentro de la ética del lenguaje) de no usar (como norma) esta característica. Ya tendremos ocasión de discutir éste y otros puntos cuando examinemos la semántica que se oculta tras el mecanismo formal de derivación.

Vemos en el código, también, que se han realizado sendos *cast*s (todavía no sabemos nada de conversiones implícitas) señalando expresamente la conversión ... ¿de qué? ¿del tipo del objeto apuntado? ¡No, por supuesto! Únicamente del puntero que apunta al objeto, de tipo invariante, para que pueda producirse la asignación. Esta conversión expresa (de un puntero o referencia de una clase derivada a un puntero o referencia de una de sus clases base, directas o indirectas) es siempre segura: las clases derivadas siempre poseen una porción de sus clases base. La conversión inversa no es segura: no se puede asegurar que una clase base contenga (arriba o abajo, pues el posicionamiento depende de la implementación) una porción de clase derivada (todos los *Profesional*es son *Persona*s, pero no todas las *Persona*s son *Profesional*es), de tal manera que esa codifificación supone un metaconocimiento de la estructura del programa, propio del desarrollador, favorecedor de errores y dependencias sin evidencia documental para el cliente de las clases.

¡Vaya! Entonces, y con todo, ¿es así de simple? -preguntaría aquí el lector- ¿Qué pasa, pues, con los accesos bajo derivación? ¿No influyen aquí? ¡Naturalmente que sí! El lector, como siempre, ha puesto el dedo en la llaga. Recordemos que la derivación pública, aplicable a las clases notadas en el código, no impone en las clases derivadas ninguna restricción de acceso sobre los miembros de las clases base. Podríamos decir, así, que las porciones del objeto correspondientes a las clases bases públicas son "públicas" para éste, entendiendo aquí el termino "público" en un sentido extensivo más que como mera etiqueta de acceso, exactamente igual que si dijéramos que la totalidad de los miembros de una clase son "públicos" con

respecto de una función miembro o amiga. De hecho, vamos a considerar, antes de seguir, un nuevo punto de vista, más general, sobre los niveles de acceso en jerarquías interrelacionadas de clases. Dejemos a un lado las etiquetas de acceso incluidas en la definición del lenguaje e, intentando que el lector olvide lo hasta ahora aprendido, califiquemos el acceso desde un ámbito a una función o variable cualquiera, a la brava, simplemente de la siguiente forma (y aquí las mayúsculas aportarán un nivel de distinción importante para el lector): *PÚBLICO* si se puede acceder, *PRIVADO* si no se puede. Bajo esta nueva luz, reconsideremos ahora lo ya visto:

- La sección *private* de una clase es PÚBLICA con respecto de (*en el ámbito de*) funciones miembros de la clase y funciones amigas de la clase, y es PRIVADA para el resto del programa.
- La sección *protected* de una clase es PÚBLICA con respecto de las funciones miembros y amigas de la propia clase (como la sección *private*) y también de las funciones miembros y amigas de las clases derivadas de ésta (he aquí la diferencia con la sección anterior), mientras que es PRIVADA para el resto del programa.
- La sección *public* de una clase es PÚBLICA para todo el programa.

Y ahora ya estamos preparados para enunciar una regla fundamental: cuando en un determinado ámbito se da una relación de derivación con respecto a una clase base dada, con cualificación de acceso PÚBLICA, entonces son de aplicación las siguientes conversiones estándar:

- un puntero a la clase derivada se convierte implícitamente en un puntero a la clase base de acceso PÚBLICO en tal ámbito.
- una referencia a la clase derivada se convierte implícitamente en una referencia a la clase base de acceso PÚBLICO en tal ámbito.
- un objeto de la clase derivada se convierte implícitamente en un objeto de la clase base de acceso PÚBLICO en tal ámbito.
- un puntero a un miembro de la clase base, de acceso PÚBLICO en tal ámbito, se convierte implícitamente en un puntero a un miembro de la clase derivada.

Por supuesto una derivación *public* equivale a un acceso PÚBLICO en cualquier ámbito. ¡Alto! ¡Un momento! ¿No conserva la derivación pública los niveles de acceso de la clase base? ¿No habrá, así, secciones en una derivación pública inaccesibles para los objetos? ¡Naturalmente! Al decir que una derivación *public* supone un acceso PÚBLICO estoy únicamente afirmando que tal tipo de derivación **posibilita** el acceso, en todo ámbito, a la porción de la clase base. Naturalmente, también, si tal clase base no dispusiera, por ejemplo, de sección *public*, los objetos de las clases derivadas de la misma (y también los objetos de la misma clase base) no podrían acceder directamente a ningún miembro de esa clase base. Veamos otro ejemplo exactamente a la inversa: si, verbigratia, la clase base no tuviera secciones *private* o *protected*, el acceso desde la clase derivada no observaría ninguna restricción real. La verdad es que usar los mismos

términos para designar distintos conceptos no es, ciertamente, algo que ayude a los principiantes en el aprendizaje del lenguaje. En realidad, donde aparece PÚBLICO debería decir "DE ACCESO POSIBLE", y donde aparece PRIVATE debería aparecer "DE ACCESO IMPOSIBLE". Pero lo cierto es que en todos los textos (ingleses, claro) sobre C++ se emplea profusamente la ambigua notación expuesta (sin distinguir entre mayúsuculas y minúsculas), esperando de la inteligencia del lector el instantáneo reconocimiento del contexto que cualifica uno u otro significado. Así las cosas y para no acostumbrar al lector a facilidades que en ningún otro texto va a encontrar, en adelante hablaremos de accesos públicos o privados, sin más.

Bien: decíamos que, en todo ámbito, los punteros y referencias a objetos de una clase derivada se convierten implícitamente en punteros y referencias, respectivamente, a objetos de la clase de la que **públicamente** se derivan. Los objetos de la clase derivada se convierten implícitamente, también, en objetos de la clase de la que públicamente se deriva aquélla, etc. etc. De esta forma podríamos codificar:

```
Directivo* punteroADirectivo = new Directivo(      "Emilio",
                                                                    2, 40000 );
Profesional* punteroAProfesional = punteroADirectivo;
Persona& referenciaAPersona = *punteroADirectivo;
Profesional encargado( "Enrique", 20 );
Persona marido = encargado;
```

Note el lector que aquí no aparecen *casts*, pues con arreglo a lo expuesto las conversiones codificadas se pueden producir de forma implícita.

Pero también existen otros ámbitos, en derivaciones no-públicas, donde se dan accesos PÚBLICOS (*lisez* "posibilidades de acceso") a las clases base. Imaginemos pues, con fines didácticos, el siguiente esquema derivativo:

```
class Base { /* ... */ };
class DerivadaProtegida : protected Base { /* ... */ };
class DerivadaProtegidaProtegida
        : protected DerivadaProtegida {
// cualquier etiqueta de acceso
        void unaFuncion(    DerivadaProtegida* pObjetoD,
                                DerivadaProtegidaProtegida* pObjetoDD )
        {
                Base* punteroABase = pObjetoD;        // OK (Atención:
                                                                    // ver nota 26 )
                punteroABase = pObjetoDD;  // OK (Atención:
                                                            // ver nota 26 )
                pObjetoD = pObjetoDD;        // OK: conversión implícita
                // ...
        }
        // ...
};
```

¿Qué ocurre aquí? ¿Realmente en el ámbito de definición de *unaFuncion* se tiene acceso público a las clase bases directa (*DerivadaProtegida*) e indirecta (*Base*), siendo ambas clases base *protected*? Bueno, si el lector quiere convencerse sólo tiene que comprobar lo siguiente: ¿se puede acceder desde tal ámbito a las secciones *public* y *protected* de las clases base? ¡Naturalmente que sí, en este caso! Sí, desde una clase derivada a la clase base directa de la que esta se deriva de forma protegida, pues recordemos que lo que así se consigue es transformar las secciones *public* y *protected* de la clase base en secciones *protected* de la clase derivada, y, por supuesto, una función miembro de una clase cualquiera tiene acceso (PÚBLICO) a todas sus secciones! Sí, también, con respecto de la clase base indirecta, pues al derivar *DerivadaProtegida* de ésta de forma protegida, las secciones no-*private* de la clase *Base* se convierten en secciones *protected* de la clase derivada, y aquí podríamos enlazar el razonamiento para con la clase base directa[26].

Bueno, revisemos ahora un ejemplo un tanto más complejo:

```
class Base { /* ... */ };
class DerPub : public Base { /* ... */ };
class DerProtPub : protected DerPub { /* ... */ };
class DerPrivProtPub : private DerProtPub {
// cualquier etiqueta de acceso
        void otraFuncion(   Base* pB, DerPub* pDP,
                                        DerProtPub* pDPP,
                                        DerPrivProtPub* pDPPP )
        {
                pB = pDP;               // OK: Base es una clase base
                                        //  pública de DerPub
                pB = pDPP;              // OK (Atención: ver nota 26 )
                pB = pDPPP;             // OK (Atención: ver nota 26 )
                pDP = pDPP;             // OK (Atención: ver nota 26 )
                pDP = pDPPP;    // OK (Atención: ver nota 26 )
                pDPP = pDPPP;   // OK: las funciones amigas y
                                        //  miembros de una clase derivada
                                        //  (cual es la presente)
                                        //  privadamente de otra
                                        //  tienen acceso público
                                        //  a la clase base
        }
        // ...
};
```

[26] Entramos de nuevo en el espinoso tema de las implementaciones particulares de C++: aun cuando lo expuesto es correcto y lógicamente impecable, si intentamos compilar los ejemplos con, verbigratia, Borland C++ 4.0, algunas líneas serán flageladas como errores del tipo "No se puede convertir X* a Y*". Concretamente el compilador encontrará problemas, normalmente, con los niveles de acceso a las clases base indirectas. Estamos, simplemente, ante una limitación del compilador.

```
funcionGlobal(      Base* pB, DerPub* pDP,
                        DerProtPub* pDPP, DerPrivProtPub* pDPPP )
{
                    pB = pDP;                   // OK: Base es una clase base
                                                //  pública de DerPub
                    pB = pDPP;                  // Error
                    pB = pDPPP;                 // Error
                    pDP = pDPP;                 // Error
                    pDP = pDPPP;        // Error
                    pDPP = pDPPP;       // Error
}
```

Bueno, a pesar de lo expuesto, yo me daría por satisfecho si el lector retuviera solamente la siguiente idea parcial: si una clase D deriva públicamente, directa o indirectamente, de otra B, dondequiera que aparezca un objeto B o un puntero o referencia a un objeto B, se puede aplicar un objeto D, o un puntero o referencia a un objeto D, respectivamente. De esta manera se quiere indicar que un D **es un** B. Bueno, veámoslo con *antropolook*: *Profesional* deriva públicamente de *Persona*, lo que quiere decir, informalmente, que un profesional es una pesona, y, por tanto, donde y como quiera que actúe una persona puede actuar, siempre, un profesional: si una persona puede imprimir su edad, un profesional también. Lo mismo ocurriría con la derivación pública de *Directivo* con respecto de *Profesional*. Claro que el razonamiento inverso no siempre funciona: o sea, un persona no tiene por qué ser un profesional, pues este tiene características (como miembros que tendría la clase) que no son de aplicación a aquélla (como los años de ejercicio).

De acuerdo: ya hemos visto las conversiones estándar, pero ¿de qué puede servir, en la práctica, todo esto? Bueno, se espera que nos habrá de simplificar la vida, permitiéndonos montar codificaciones "genéricas" como la siguiente:

```
void comoTeLlamas( const Profesional& individuo )
{
        //  un objeto Profesional tiene acceso a la
        // sección pública de la clase Persona
        cout << Profesional.nombre();
}

Profesional unProfesional( "Emilio", 2 );
comoTeLlamas( unProfesional );          // OK: imprime "Emilio"
Directivo unDirectivo( "Don Emilio", 2, 50000 );
// un Directivo es un Profesional,
//  y en la siguiente línea se produce
//  una conversión implícita de Directivo a Profesional
comoTeLlamas( unDirectivo ); // OK: imprime "Don Emilio"
```

Si el lector no ha asimilado del todo lo hasta ahora expuesto, al menos habrá retenido un concepto: "los miembros de la sección privada de una clase no se pueden acceden desde fuera del protocolo de descripción de su propia clase (habiendo abstracción de las funciones amigas, un mal necesario que debemos evitar en lo posible)". La idea que predomina en la orientación a objetos de C++ es, por otro lado, que "cambios pequeños originan pequeñas repercusiones". Imaginemos ahora una clase insertada en una jerarquía, con su código utilizado y re-utilizado por otras clases y objetos: los miembros de la sección *private* de nuestra clase no formarán parte del interfaz utilizado por los clientes de la misma. Si modificamos, no obstante, tal sección *private*, inmersa en un archivo de cabecera donde se ha definido completamente la clase, al cambiar la fecha adscrita al archivo de cabecera nuestra utilidad *make* forazará la recompilación de nuestro programa: ¡esto no es lógico! De alguna manera tendríamos que convertir la barrera lógica de restricción de acceso a la sección *private* en una barrera formal que impidiera estas no-deseadas consecuencias. Una buena idea a este respecto es la aportada por John Carolan en su trabajo "Construyendo clases a prueba de balas". Se trata, en definitiva, de la técnica del "Gato de Cheshire" (y aquí los lectores de Carroll recordarán las voluntarias desapariciones parciales del risueño animal). Es como si dijéramos: aquí está la cola del gato, ¿dónde está el cuerpo? La técnica consiste, pues, en sustituir la sección *private* de una clase por un puntero a otra clase, con acceso total *private*, donde se implemente el detalle de la representación interna de la primera y se declare éste como amiga. ¿Un poco oscuro? Veámoslo en la práctica: por un lado tenemos un fichero de cabecera donde aparece el siguiente código:

```
class Persona;

class RepInternaPersona {     // el cuerpo del gato de Cheshire
friend class Persona;                    // la situación de la cola
                                                  //  de nuestro gato.
private:
        RepInternaPersona() {
                nombre_ = new char[ 1 ];
                nombre = '\0';
        }
        RepInternaPersona( char* unNombre ) {
                nombre = new char[ strlen( unNombre ) + 1 ];
                strcpy( nombre_, unNombre );
        }
        ~RepInternaPersona() {
                delete [] nombre_;
        }
private:
        char* nombre_;
};
```

Por otro lado tenemos el fichero en el que definimos nuestra clase *Persona*:

```
class RepInternaPersona;

class Persona {
public:
        char* nombre() const {
                return rip->nombre_;
        }
protected:
        Persona() {
                rip = new RepInternaPersona;
        }
        Persona( char* unNombre ) {
                rip = new RepInternaPersona( unNombre );
        }
        ~Persona() {
                delete rip;
        }
private:
        RepInternaPersona* rip;        // la cola del gato de Cheshire
};
```

Francamente, el código escrito no es demasiado eficiente (ni siquiera en el tratamiento de los constructores), pero mirémoslo desde este punto de vista: modificar la representación privada de nuestra clase *Persona* es modificar la clase *RepInternaPersona*, contenida en un fichero distinto, que se convierte así en un "cortafuegos" a efectos de recompilación.

Bien, este es el momento en que el lector preguntará: ¿a qué han venido estos fuegos de artificio? Y la respuesta es ... ¡ea! ¡no podemos dejar que el lector piense que C++ es un lenguaje fácil o lineal! Como he indicado en repetidas ocasiones, en C++ los límites se amplían y modifican día a día: C++ es un lenguaje poblado de distintos idiomas, según la terminología ya clásica de Jim Coplien, entendiendo por idiomas la conjunción de ciertos aproximaciones teóricas y técnicas que permiten unos esquemas lógicos y formales de codificación notablemente diferenciados entre sí. Así, al igual que existe un "idioma de constructores virtuales" o un "idioma de ejemplares" o un "idioma tipo Smalltalk", también existe un idioma que podríamos llamar de "modulación protectiva en compilación", con sus lógicas ventajas y desventajas. Esta flexibilidad, para algunos malsana, es una característica que a nadie deja indiferente.

10

DE LAS FUNCIONES VIRTUALES

Conocemos que la herencia se implementa en C++ mediante la derivación de clases; hemos aprendido, también, los formalismos sintácticos para la aplicación de tal mecanismo; sabemos, al fin, de conversiones implícitas en derivación y del orden en la aplicación de constructores y destructores. ¿Y ahora qué? Bien, esa es exactamente la cuestión: ahora entra en escena la estrella, el epicentro de una nueva forma de programar: la *función virtual*. Pero no adelantemos acontecimientos: seamos objetivos.

REUTILIZACIÓN Y REFINAMIENTO DEL CÓDIGO HEREDADO

Como ya vimos en el capítulo anterior, el mecanismo de derivación pública de clases nos permite implementar codificaciones "genéricas" mediante una sintaxis que los programadores de C podrían fácilmente asociar, respetado el debido distanciamiento, con la técnica de uso de punteros a void. De esta forma, si tenemos la siguiente jerarquía de clases:

```
class Empleado {
public:
        int antiguedadEnLaEmpresa()
        {
                return antiguedad;
        }
        void estableceAntiguedad( int anos )
        {
                antiguedad = anos;
        }
private:
        int antiguedad;
// ...
};
class JefeDeSeccion : public Empleado { /* ... */ };
class Administrativo : public Empleado { /* ... */ };
class DirectorGeneral : public Empleado { /* ... */ };
// etc., etc.
```

podemos codificar una función de uso "parcialmente genérico" tal como

```
void imprimeAntiguedad( Empleado* punteroAEmpleado )
{
        cout << punteroAEmpleado->antiguedadEnLaEmpresa();
}
```

que nos permitirá codificar fragmentos como el siguiente:

```
JefeDeSeccion* pFranklin;
Administrativo* pEleanor;
DirectorGeneral* pFrank;
// ...
imprimeAntiguedad( pFranklin );        // OK
imprimeAntiguedad( pEleanor );         // OK
imprimeAntiguedad( pFrank );           // OK
```

De esta manera se produce un doble ahorro: por un lado reutilizamos la función *imprimeAntiguedad*, y por otro reutilizamos, también para toda la jerarquía de clases, la función miembro *antiguedadEnLaEmpresa*. ¿Perfecto? Bien, esto se anima. Reutilizamos, en definitiva, en las clases derivadas (¡públicamente, claro!) cuanto tenemos en la clase base o cuanto a ésta pueda aplicarse. ¿Qué pasaría, sin embargo, si deseáramos particularizar directamente algunas de nuestras clases derivadas? Imaginemos una situación como la siguiente:

```
class DirectorGeneral : public Empleado {
public:
        char* antiguedadEnLaEmpresa() const
        {
                return "Antiguedad sin importancia";
        }
// ...
};

DirectorGeneral* pFederico = new DirectorGeneral;
pFederico->estableceAntiguedad( 3 );
imprimeAntiguedad( pFederico );
```

¿Puede el lector adivinar el resultado de la última línea de código? Parece que, dado que se ha pasado a la función un puntero a un objeto del tipo *DirectorGeneral*, deberá ejecutarse la función miembro de esta clase, que devolverá una cadena cuando, en el ámbito de nuestra función global, se ejecute la línea

```
cout << punteroAEmpleado->antiguedadEnLaEmpresa();
```

¿Correcto? ¡No! Lo cierto es que, para desesperanza del lector, el resultado es ... ¡la impresión del entero "*3*"!. ¿Qué ocurre aquí? Bueno, examinemos el código con más detenimiento. Efectivamente se pasa a nuestra función

global un puntero a un objeto de tipo *DirectorGeneral*, pero la función espera en realidad un puntero a un objeto de tipo *Empleado*, por lo que, merced a que la clase *DirectorGeneral* se deriva públicamente de la clase *Empleado*, el puntero a *DirectorGeneral* se convierte implícitamente en un puntero a *Empleado*. O sea, el puntero, una vez "convertido", permite acceder al objeto de tipo *DirectorGeneral* a través de la porción de éste correspondiente a su clase base *Empleado* o, lo que es lo mismo, el objeto apuntado por *pFederico* se trata como si fuera un objeto *Empleado*, y en estos objetos la función *antiguedadEnLaEmpresa* devuelve un entero. Pero quizá el lector lo habrá de ver más claro en el siguiente ejemplo circunstancial:

```
DirectorGeneral* punteroADirectorGeneral = new DirectorGeneral;
punteroADirectorGeneral->estableceAntiguedad( 7 );
Empleado* punteroAEmpleado = punteroADirectorGeneral;

// la siguiente línea devuelve "Antiguedad sin importancia"
punteroADirectorGeneral->antiguedadEnLaEmpresa();
punteroAEmpleado->antiguedadEnLaEmpresa(); // devuelve '7'
```

Esto es, se ejecutará la función correspondiente a la porción de la clase desde la que se acceda al objeto. Naturalmente si en la pertinente porción no existe la función, ésta se buscará en la porción de la clase de la que éste públicamente pudiera derivar, y si tampoco existe en ésta se buscará en la clase base pública de la misma, y así recursivamente. La búsqueda de miembros se realiza, pues, desde las clases derivadas hacia las clases bases, y no al revés, lo que tiene una explicación intuitiva muy clara: sabiendo que la clase de un objeto deriva públicamente de otra, si accedemos a un objeto a través de la porción de una clase derivada se sabe que existirá una porción, al menos, de la clase base en él, y en ésta se buscará seguidamente; si accedemos al objeto, en cambio, a través de la porción de la clase base, el sistema no tiene forma de determinar si tal porción corresponde a un objeto de la clase base o de las derivadas, por lo que siempre asume que el tipo del objeto corresponde a la clase base.

Observamos también, incidentalmente, un curioso y desagradable efecto que Scott Meyers denomina "*comportamiento esquizofrénico*" en los objetos de tipo *DirectorGeneral*: dependiendo desde donde se acceda a los mismos, el comportamiento de éstos será distinto. ¡Esto no es lógico! Cuidémonos, pues, de redefinir en las clases derivadas las funciones heredadas de las clases bases.

¡Vaya!, podría exclamar muy bien aquí el inquieto lector, ¡ya entiendo: si redefinimos una función heredada de una clase base en una clase derivada, lo que ocurre es que se oculta la función de la clase base! Efectivamente, inteligente lector. Pero, podría seguir el mismo lector, ¿y si en lugar de redefinir la función de la clase base lo que hacemos es sobrecargarla? ¿Tendremos entonces acceso a ambas? Vaya, cómo se nota que el lector

aprende rápido y ya plantea difíciles cuestiones. Veámoslo con un sencillo ejemplo:

```
class DirectorGeneral : public Empleado {
public:
        int antiguedadEnLaEmpresa( int* passnumber )
        {
                if ( passnumber == 17 )
                        return antiguedad;
                else
                        return -1;
        }
// ...
};
```

Parece que ahora, desde un objeto de tipo *DirectorGeneral*, podríamos acceder tanto a la función sin argumentos como a la función con un argumento entero. Parece, parece, pero NO es:

```
DirectorGeneral miJefe;
miJefe.estableceAntiguedad( 5 );
miJefe.antiguedadEnLaEmpresa( 17 );   // OK: devuelve '5'
miJefe.antiguedadEnLaEmpresa(12 );    // OK: devuelve '-1'
// seguidamente intentamos acceder a la
//  función heredada de la clase base
miJefe.antiguedadEnLaEmpresa();       // ERROR
```

¿Qué ocurre, de nuevo? ¡Pues que no hay tal sobrecarga! Las funciones se han declarado en diferentes ámbitos, por lo que el nombre de la función con un argumento esconde o "tapa" el nombre de la función sin argumentos. Si deseáramos todavía, empero, realizar esta sobrecarga lo que deberíamos hacer es declarar ambas funciones en el mismo ámbito, añadiendo la siguiente función a la clase derivada:

```
inline int DirectorGeneral::antiguedadEnLaEmpresa()
{
        Empleado::antiguedadEnLaEmpresa();
}
```

Vemos, pues, que, aparte del comportamiento esquizofrénico antes notado, si redefinimos en una clase derivada una función de una clase base, estaremos ocultando la función de la clase base *y todas las sobrecargas de la misma en la clase base*.

SELECTORES DE TIPO

Pero volvamos al origen de nuestra disquisición: lo que pretendíamos era poder usar de una codificación genérica pero que, a la vez, nos permitiera particularizar determinados métodos para ciertas clases. Hemos visto que esto no lo podemos hacer mediante punteros a la porción de la clase base de

los objetos, pues así únicamente accederíamos a la función común de la clase base misma. Pero, ¿y si realizáramos un *cast* expreso al puntero para que el objeto fuera accedido desde la porción de clase adecuada? ¡Perfecto! Pero, ¿cómo saber qué *cast* aplicar en cada caso? O sea, ¿como podemos saber, en el ámbito de una función genérica para una determinada jerarquía, a qué clase de objeto pertenece el objeto apuntado por un puntero ya convertido a puntero a la clase base? Bueno, podríamos conseguirlo añadiendo a nuestras clases lo que se denomina un "campo selector de tipo". Se trata, en definitiva, de un nuevo campo, común a todas nuestras clases (y por tanto candidato perfecto para la clase base), que tomará un valor particular para cada clase, lo que nos permitirá reconocer de qué clase es cada objeto. En primer lugar, pues, veremos los cambios en la clase base:

```
class Empleado {
public:
        enum    tipoEmpleado{ JEFESEC, ADMTVO, DTORGEN }
                tipoDeEmpleado() {
                        return valorTipoEmpleado;
        }
        char* lema() const
        {       // lemas de las distintas categorías de empleados
                // para ser particularizados en las clases derivadas
                return "Los empleados son el alma de la empresa";
        }
        // …
private:
        tipoEmpleado valorTipoEmpleado;
        // …
};
```

Seguidamente tendríamos que dotar a las clases derivadas de unos constructores apropiados que inicialicen debidamente el campo selector de tipo. Obviaremos, a este concreto fin, el cuerpo de los constructores, resultando en algo así como lo siguiente:

```
class Administrativo : public Empleado {
public:
        Administrativo()    : valorTipoEmpleado( ADMTVO ) ...
                                // sigue lista inicialización
        {
                // aquí viene el cuerpo del constructor
        };
        Administrativo( Administrativo& unAdmtvo )
                : valorTipoEmpleado( ADMTVO ) ...
        {
                // cuerpo del constructor de copia
        }
        char* lema() const
        {
                return "Los Administrativos sostienen a la empresa";
        }
        // …
```

```
};

class DirectorGeneral : public Empleado {
public:
        DirectorGeneral()
                : valorTipoEmpleado( DTORGEN ) ... { /* ... */ }
        DirectorGeneral( DirectorGeneral& miDG )
                : valorTipoEmpleado( DTORGEN ) ... { /* ... */ }
        char* lema() const {
        return "El Director General es el corazón de la empresa";
        }
        // ...
};

class JefeDeSeccion : public Empleado {
public:
        JefeDeSeccion()
                : valorTipoEmpleado( JEFESEC ) ... { /* ... */ }
        JefeDeSeccion( JefeDeSeccion& miJDS )
                : valorTipoEmpleado( JEFESEC ) ... { /* ... */ }
        char* lema() const {
return "Los Jefes de Sección conducen realmente la empresa";
        }
        // ...
};
```

Ahora codifiquemos nuestra función "genérica":

```
void imprimeIdeario( Empleado* pEmpleado )
{
        switch( pEmpleado->tipoDeEmpleado() ) {
        case ADMTVO:
                cout    << (Administrativo* )pEmpleado->lema()
                        << '\n';
                break;
        case JEFESEC:
                cout    << (JefeDeSeccion* )pEmpleado->lema()
                        << '\n';
                break;
        case DTORGEN:
                cout    << ( DirectorGeneral* )pEmpleado->lema()
                        << '\n';
                break;
        default:
                cout << pEmpleado->lema() << '\n';
}
```

de forma que las siguientes líneas:

```
Empleado* juanNadie;
Administrativo* ruperez;
JefeDeSeccion* carvajal;
DirectorGeneral* deLasHeras;
// ...
```

```
imprimeIdeario( juanNadie );
imprimeIdeario( ruperez );
imprimeIdeario( carvajal );
imprimeIdeario( deLasHeras );
```

originarán la siguiente salida:

```
Los Empleados son el alma de la empresa
Los Administrativos sostienen a la empresa
Los Jefes de Sección conducen realmente la empresa
El Director General es el corazón de la empresa
```

Por supuesto que las particularizaciones podrían haberse realizado también con la intervención de la representación interna de los objetos, pero vayamos a la esencia de lo expuesto: de una manera similar a la bien conocida por los programadores de C, hemos creado una función que puede "identificar" en tiempo de ejecución los objetos de distintos tipos (en jerarquía de clases) que se les pasen como argumentos, pero el sistema observa algunas graves deficiencias y peligros. Pensemos, si no, en que, por ejemplo, ya tuviéramos una buena cantidad de funciones así codificadas (con la aburrida letanía de código repetido que esto supone) y que, por exigencias del problema en la vida real, debiéramos anexar una nueva clase derivada a nuestra jerarquía: tendríamos, primero, que añadir un nuevo enumerador a la clase *Empleado*; seguidamente deberíamos adicionar inicializadores para tal enumerador en los correspondientes constructores de la clase derivada al efecto; habría que modificar también, por último, todas las funciones "genéricas" aplicándoles un nuevo bloque *case*. Como el lector ya habrá adivinado, este proceso puede llegar a ser muy complicado, enturbiando, de cualquier forma, la mantenibilidad del código. Y desde luego que no es esto lo que se espera de C++.

FUNCIONES VIRTUALES

¿No habría alguna manera de que fuera el mismo sistema el encargado de la identificación, en tiempo de ejecución, de los objetos? O, lo que es lo mismo, ¿no podría el sistema generar por sí mismo el código necesario, sobre la base conceptual expuesta, para procurar la identificación necesaria para así llamar a la función apropiada? Bien, sí podría. De hecho esta capacidad se denomina *identificación de operadores en tiempo de ejecución*, y junto con el mecanismo de derivación, permite en C++ la tan cacareada Programación Orientada-a-Objetos.

El mecanismo que usa C++ para procurar tal servicio es la *función virtual*. En definitiva se trata de lo siguiente: si sabemos que una función de una clase base va a ser particularizada o redefinida en una o más clases derivadas, esto se lo haremos saber al compilador notándole que se trata de una función *virtual*. Haciéndolo así, el compilador generará para la clase código adicional (algo así como el selector de tipo visto anteriormente) que

permitirá la asociación de un objeto con la función (particularizada o no) que le corresponda. ¡Un momento, un momento! No estamos hablando aquí de que el lenguaje C++ imbuya un método de reconocimiento en tiempo de ejecución del tipo de los objetos. De hecho, la identificación del tipo de los objetos en tiempo de ejecución es un tema actualmente en discusión, basado en un trabajo recientemente presentado por Bjarne Stroustrup y Dmitri Lenkov a X3J16. O sea, el lenguaje no proporciona mecanismos (todavía) para que un objeto pueda responder a la pregunta: ¿de qué tipo eres? Naturalmente podemos simularlo artificialmente, como ya hemos visto, pero lo cierto es que en la comunidad C++ casi se ha instaurado como norma el lema: "no es de buen estilo usar campos selectores de tipo en el diseño de clases"[27]. Así están las cosas. ¡Vaya! Entonces, si las funciones virtuales no proporcionan la identificación del tipo de los objetos en tiempo de ejecución, ¿qué demonios hacen? Veámoslo desde el punto de vista de Orientación-a-Objetos: si recabamos en que un sistema software está constituido por objetos y sus interrelaciones (los mensajes), vemos que normalmente un mensaje dirigido a un objeto (la signatura de una función, para entendernos) se asocia con un método (una función miembro) en tiempo de compilación. Esto se denomina *ligadura estática*, y permite la optimización del código, pues el compilador se asegura, mediante el chequeo estático de tipos, que el objeto a que se dirige el mensaje contiene el método apropiado para contestarlo, señalando un error en caso contrario. Cuando estamos trabajando con clases con funciones miembros "normales", el compilador no se preocupa de más que de lo evidente. Así, en el ejemplo anterior:

```
void imprimeAntiguedad( Empleado* punteroAEmpleado )
{
        cout << punteroAEmpleado->antiguedadEnLaEmpresa();
}
```

el sistema asocia, en el momento de la compilación, el mensaje *antiguedadEnLaEmpresa* al método del mismo nombre... ¡en la clase *Empleado*!. El sistema no quiere saber nada, a efectos de la resolución de mensajes, de objetos de otros tipos, cuyas clases derivan de *Empleado*, pues nada de ellos aparece en el código expuesto: el mensaje se dirige a un objeto *Empleado* a través de un puntero al mismo, y punto. Pero, claro, no es esto lo que queremos. Sigamos adelante, pues.

[27] En el diseño del lenguaje C++ se sopesó sobremanera la posibilidad de incluir una tal identificación en las clases, pero se estimó que tal posibilidad abocaría con facilidad las codificaciones hacia estructuras de tipo *switch*, perjudicando la modularidad y mantenibilidad de los programas. De hecho, las deficiencias observadas en el lenguaje SIMULA derivadas de tal identificación de tipos, que este lenguaje sí contempla, influyeron decisivamente en la decisión de no incluirlos en C++.

Existe, como el lector ya habrá adivinado, otro método de resolución de mensajes, denominado *ligadura dinámica*, que permite, en esencia, que un mensaje pueda ser asociado con el método de un determinado objeto en tiempo de ejecución. Esto se consigue en C++ si los métodos a asociar son *virtuales*. O sea, y volviendo al ejemplo anterior, si el método *antiguedadEnLaEmpresa* fuera *virtual* (a conseguir mediante una sencilla sintaxis, como inmediatamente veremos), al sistema se le está diciendo que codifique cierta información al construir los objetos de las clases que contengan tal método, de manera que sea él mismo el que se ocupe de averiguar en qué clase debe buscar el método (particularizado o no) debidamente asociado al objeto, accedido bien mediante un puntero bien mediante una referencia, que se pasa como argumento.

Bien, ¿cómo se convierte un método "normal" en "*virtual*"? Pues fácilmente: antecediendo la declaración de la función miembro en la clase base por la palabra clase **virtual**. Volviendo a nuestro ejemplo de los lemas, lo único que tendríamos que hacer es lo siguiente:

```
class Empleado {
public:
        virtual char* lema() const
        {
                return "Los empleados son el alma de la empresa";
        }
        // …
};
```

Con esta simple adición (y una vez suprimido el código generado por los campos selectores de tipo), nuestra función "genérica" podría quedar simplemente así:

```
void imprimeIdeario( Empleado* pEmpleado )
{
        cout << pEmpleado->lema() << '\n';
}
```

de forma que las siguientes líneas:

```
Empleado* juanNadie;
Administrativo* ruperez;
JefeDeSeccion* carvajal;
DirectorGeneral* deLasHeras;
// …
imprimeIdeario( juanNadie );
imprimeIdeario( ruperez );
imprimeIdeario( carvajal );
imprimeIdeario( deLasHeras );
```

originarán la siguiente salida:

```
Los Empleados son el alma de la empresa
```

```
Los Administrativos sostienen a la empresa
Los Jefes de Sección conducen realmente la empresa
El Director General es el corazón de la empresa
```

que es exactamente lo que deseábamos. ¡Vaya! Parece que ya se empieza a ver el oro bajo las plomizas dificultades sintáticas de C++: el código de la función se nos ha acortado drásticamente. Pero, ¿qué pasaría si tuviéramos que añadir a nuestra jerarquía una nueva clase con una redefinición de nuestra función virtual? ¡Nada que nos deba preocupar! Veámoslo:

```
class Mensajero : public Empleado {
public:
        char* lema() const {
        return "Los Mensajeros son las arterias de la empresa";
        }
        // …
};

Mensajero* juan;
// …
imprimeIdeario( juan );
```

¿Qué imprimirá la última línea? Pues simplemente

```
Los Mensajeros son las arterias de la empresa
```

Notamos, en principio, que al declarar una función de una clase base como *virtual*, automáticamente las funciones con la misma signatura en las clases derivadas pasan a ser virtuales. De esta forma la mantenibilidad del código se simplifica sobremanera.

En resumen: una función virtual es una función miembro (no puede ser global) antecedida por la clave *virtual* en el protocolo de descripción de su clase, y tiene sentido cuando de la clase en la que se declara se derivarán otras. La palabra clave *virtual* puede usarse en la declaración de las funciones virtuales en las clases derivadas, pero es redundante. Si una función virtual no se redefine en una clase derivada, se asumirá para ésta la definición de tal función en su clase base. Una función virtual puede ser declarada *inline*, pero esta declaración únicamente deberá tener efecto cuando la llamada a tal función se resuelva, como es lógico, en tiempo de compilación.

De acuerdo: esto parece funcionar, pero ¿cuál es el mecanismo que subyace bajo esta característica del lenguaje? Intentaré explicarlo: cuando el compilador se encuentra con una clase que contiene una función virtual se añaden al código de la clase dada y de sus clases derivadas sendos punteros a un array de punteros a funciones denominado *tabla de funciones virtuales* o *v-table*, de manera que cualquier llamada a una función virtual se resolverá mediante un nivel adicional de indirección: el paso por el array de funciones virtuales. Este puntero incrementará el tamaño de cada objeto, por

un lado, a la vez que el mecanismo de indirección, por otro, originará una cierta penalización en la ejecución de la llamada a la función virtual: pequeñas (y ya veremos por qué) desventajas para una gran facilidad. La *v-table* poseerá una entrada por cada una de las funciones virtuales definidas en la jerarquía de clases en cuestión, y se alojará en un espacio de la clase inaccesible al usuario, normalmente en una locación estática para permitir su acceso a todos los objetos de la clase. En el ejemplo de empleados que estamos tratanto, cada una de las distintas clases de la jerarquía construiría su propia tabla virtual, añadiéndose a las clases el código necesario para inicializar debidamente el puntero *vptr* de forma que apuntara a la *v-table* apropiada. Como vemos, el sustrato conceptual es similar al usado con los campos selectores de tipo.

CUESTIONES DE ÁMBITO EN FUNCIONES VIRTUALES

Cuando trocamos una función miembro en *virtual* debemos recabar en que algunos aspectos relacionados con la misma han cambiado.

Es conveniente que pensemos que la cualificación de *virtual* se refiere a un conjunto de funciones, más que a una sola, compuesto por la declarada expresamente como virtual y las que, con la misma signatura, pertenecen a las clases derivadas públicamente de la clase que alberga a la primera. De alguna forma debemos pensar en una única "plantilla" de función repartida entre diferentes clases de una jerarquía: el sistema no sabe, en tiempo de compilación, a qué clase, incluyendo una función miembro virtual, pertenecerá un determinado objeto (no conoce, en definitiva, el cuerpo concreto de la función a ejecutar), pero es evidente que debe conocer con exactitud el prototipo exacto de la función virtual. Esto es, pasando al plano de objetos, el mensaje es el mismo y únicamente cambia el método, porque si las funciones virtuales tuvieran distintos prototipos en las distintas clases a las que pertenecen, ¿cómo podríamos codificar una llamada genérica a las mismas? Lancémonos a un ejemplo: hemos declarado *virtual* la función miembro *lema()* en la clase *Empleado* con el siguiente prototipo:

```
class Empleado {
public:
        virtual char* lema() const {
                // cuerpo de la función
        }
        // …
};
```

Siempre que usemos tal función (o mejor "grupo de funciones") en nuestro código trataremos con una función sin argumentos, de nombre *lema*, que devuelve un puntero a *char*. Imaginemos ahora que una determinada clase desea extender la particularización de la función a su prototipo:

```
class Escribiente : public Empleado {
```

```
public:
        char* lema( TipoDeLetra fuente )
        {       // se pasa como argumento un objeto
                //  del tipo TipoDeLetra
                // imprime el lema con el tipo de letra "fuente"
        }
        // …
};
```

Si recordamos el cuerpo de nuestra función genérica

```
void imprimeIdeario( Empleado* pEmpleado )
{
        cout << pEmpleado->lema() << '\n';
}
```

preguntémonos ahora: ¿cómo demonios esa función sin argumentos va a llamar a una función con un argumento? Veamos qué pasa:

```
Empleado* cualquiera;
Empleado* unEmpleado = new Escribiente;
Escribiente* unEscribiente = new Escribiente;
// …
imprimeIdeario( cualquiera );  // salida: Los Empleados son
                                        //  el alma de la empresa
imprimeIdeario( unEmpleado );          // salida: Los Empleados son
                                        //  el alma de la empresa
//
imprimeIdeario( unEscribiente );       // salida: Los Empleados son
                                                //  el alma de la empresa
TipoDeLetra courier;
// …
unEscribiente->lema( courier );        // salida: Los Escribientes
                                                //  son la letra de la empresa
unEmpleado->lema( courier );           // error: no existe tal función
                                        //  en la clase Empleado
```

De igual manera como la "aparente sobrecarga" de una función no-virtual (así denominaremos en adelante a las funciones miembros "normales") de una clase base en una clase derivada origina el ocultamiento de la función de la clase base y de sus sobrecargas, así ocurre con las funciones virtuales. Si examinamos el código notamos que la función *lema(TipoDeLetra)* de la clase *Escribiente* es tratada como una función no-virtual, de tal forma que, como ya sabemos, al ser accedido el objeto mediante un puntero a su clase base, se ejecuta la función contenida en ésta. Al cambiar, pues, el prototipo de la función virtual en una clase derivada se ocultan las funciones con el mismo nombre de la clase base, que son las que detentarían la cualificación de *virtuales*, por lo que tales no serían accesibles en el ámbito de la clase derivada.

¿Qué pasaría, sin embargo -podría inquirir el astuto lector-, si declaráramos nuestra función de prototipo modificado como *virtual* en la clase derivada? Pues que en nada cambiaría lo dicho: el nombre de tal función ocultaría el de la función virtual de su clase base, aunque podría aplicarse también como virtual con efecto en las clases derivadas de, en nuestro caso, la clase *Escribiente*. Habría que tener en cuenta, no obstante, que si en las clases derivadas de nuestra clase derivada se redefine la función virtual de la clase base, ésta ocultará la función virtual de la clase derivada. ¿Complicado? Bueno, este galimatías linguístico podría expresarse así:

```
#define nl '\n'
#include <iostream.h>

class Base {
public:
	virtual void f()
	{
		cout << "Base" << nl;
	}
};

class Media : public Base {
public:
	virtual int f( int numero )
	{	// oculta la función virtual Base::f()
		cout << "Media" << nl;
		return numero;
	}
};

class Derivada : public Media {
private:
	void f()
	{	// oculta la función virtual Media::f(int)
		cout << "Derivada" << nl;
	}
};


void g( Base* puntero )
{
	puntero->f();
}

int main( int, char** )
{
	Base* pBase = new Base;
	Base* pBaseMedia = new Media;
	Base* pBaseMediaDerivada = new Derivada;

	Media* pMedia = new Media;
	Media* pMediaDerivada = new Derivada;
```

```
        Derivada* pDerivada = new Derivada;

        g( pBase );
        g( pBaseMedia );
        g( pBaseMediaDerivada );

        g( pMedia );
        g( pMediaDerivada );

        g( pDerivada );

        return 0;
}
```

¿Adivina el lector la salida de este sencillo programita? Vamos a ella:

```
Base
Base
Derivada
Base
Derivada
Derivada
```

¡Naturalmente! Es como si nuestra función virtual sin argumentos se hubiera escondido, como el Guadiana, a su paso por la clase *Media*, para volver a aparecer un poco más alla en la escala de derivación: en nuestra clase *Derivada*. Notemos que en esta última clase parece que podría haber conflicto con el nombre de dos funciones virtuales distintas, pero la cuestión se salda a favor de la función virtual con más profundidad en la escala derivativa.

Queda establecido, pues, que las funciones virtuales redefinidas en las clases derivadas de aquélla en la que se declara como virtual la primera, deben compartir exactamente la misma signatura, lo que incluye el nombre de la función, el número, tipo y orden de sus argumentos, y su tipo de retorno. Queda entendido, también, que una modificación de la signatura no constituye una sobrecarga de la función virtual. Pero, ¿y si lo que se cambia es únicamente el tipo de retorno de la función virtual? Simplemente se genera un error en compilación. Algo ási como:

```
class Media : public Base {
public:
        int f() {    // ERROR: el tipo de retorno no coincide
                return 0;
        }
};
```

CUALIFICACIÓN DE ACCESO DE LAS FUNCIONES VIRTUALES

Incidentalmente podemos notar en nuestro ejemplo que la función declarada virtual pertenecía a la sección pública de la clase *Base*, aunque su redefinición en la clase *Derivada* ha sido incluida en la sección privada de esta última clase. Hemos podido, por otra parte, acceder sin problemas a esta función virtual de la clase *Derivada* sin aparentes problemas. ¿Qué ocurre, pues, con los niveles de acceso para con las funciones virtuales? ¿Dependen únicamente de la cualificación de acceso correspondiente a la clase base en que se declarén como virtuales? ¿O sea, en nuestro caso, por ejemplo, la función virtual *f()* posee siempre acceso público por haber sido declarada *virtual* en la sección *public* de la clase *Base*? ¡No! El nivel de acceso de una función virtual queda determinado en cada caso por el nivel de acceso que posee tal función en la clase a que pertenecen los punteros o referencias a través de los que se accede a la misma. ¿Queda claro? Bueno, repasemos el ejemplo: ¿dónde se producen las invocaciones a la función virtual *f()* en el código anterior? Pues en el interior de la función *g(Base* puntero)*, concretamente en la línea

```
puntero->f();        // OK: Base::f() es pública
```

donde *puntero*, merced a la ya conocida conversión implícita de tipos entre clases derivadas públicamente, es siempre un puntero a *Base*, independientemente del tipo concreto del puntero pasado como argumento o aún del tipo del objeto apuntado por el mismo. Imaginemos lo siguiente:

```
Media* punteroADerivada = new Derivada;        // puntero a objeto
                                                                    // de clase Derivada
g( punteroADerivada);
```

En este caso el puntero originalmente es de clase *Media* y apunta a un objeto de clase *Derivada*. Sin embargo al pasarlo como argumento a la función *g(...)*, en el cuerpo de ésta, se transforma en un puntero de tipo *Base*, aunque sigue apuntando a un objeto de tipo *Derivada*. La función *f()* se accede, pues, mediante un puntero de tipo *Base**. Veamos, entonces, qué cualificación de acceso tiene la función *f()* en la clase *Base*: pública, por lo que en esta llamada de la función virtual el nivel de acceso es público. Si intentáramos acceder al mismo objeto (de tipo *Derivada*) a través, por ejemplo, de un puntero de tipo *Derivada** , como en la clase *Derivada* el nivel de accedo de la misma función virtual *f()* es *private*, el compilador flagelaría como error el siguiente código:

```
Derivada* punteroADerivada = new Derivada;
punteroADerivada->f();        // ERROR: la función Derivada::f()
                                        //  es private
```

DESTRUCTORES VIRTUALES

Volvamos a nuestro anterior jerarquía de empleados y ponderemos la siguiente situación:

```
Empleado* encargadoArchivo = new Administrativo;
// se utiliza el objeto de tipo Administrativo
//  apuntado por encargadoArchivo
// …
// seguidamente se desea destruir expresamente
//  el objeto de tipo Administrativo
delete encargadoArchivo;
```

En principio la situación parece elemental: construimos un objeto y, tras usarlo, lo desechamos llamando a su destructor mediante el operador *delete*, según la sintaxis que ya conocemos. ¿Correcto? ¡No! ¡No! Examinemos de nuevo la última línea: le pasamos un puntero al operador *delete* y suponemos que tal operador llamará al destructor del objeto, pero ¿qué es lo que realmente conoce el operador *delete* de nuestro objeto? ¡Nada! Tal operador únicamente sabe de la clase a que pertenece el puntero que se le pasa como argumento. Pero… ¡la clase del puntero es distinta a la clase del objeto que deseamos destruir! En definitiva, el compilador resuelve estáticamente la llamada al destructor, y efectivamente llama al destructor … ¡de la clase *Empleado*! Pero desde luego que no es esto lo que deseábamos. Es más: este esquema puede procurarnos más de algún problema grave. Recabemos por ejemplo, si no, en la posibilidad que el constructor de la clase *Administrativo* use de la memoria del almacenamiento libre (quizás para almacenar el nombre del empleado), ocupándose el destructor de la misma clase de liberar tal memoria. Si ejecutamos el código anterior, empero, el destructor de *Administrativo* no será llamado, con lo que tal memoria no sería liberada: ¡problema!. Pensemos por otra parte, también, en un objeto de una clase derivada de otra, y ésta de otra, y así sucesivamente hasta llegar a una clase base de cuyo tipo será el puntero direccionado a nuestro objeto. Sabemos que en una jerarquía de clases los destructores de las clases bases se ejecutan en orden inverso al de los constructores, desde las clases más derivadas hacia las clases bases. Si merced a un código similar al anterior se llama al destructor de la clase base en vez de al de la clase de nuestro objeto, se estarán obviando los destructores de las clases intermedias en la escala derivativa entrambas: ¡más problemas!

Como ya habrá adivinado el lector, tales problemas tienen solución, y singularmente fácil, a fe mía: debemos declarar los destructores como *virtuales*. O sea, debemos declarar como virtual el destructor de la clase base de la jerarquía derivativa. De esta manera el sistema, con el mismo código anterior, identificaría en tiempo de ejecución el destructor apropiado para el tipo de objeto apuntado por nuestro puntero. Montemos, sin más dilación, el ejemplo:

```
class Empleado {
public:
        Empleado() { /* ... */ }        // constructor
        virtual void f() { /* ... */ }
        // …
```

```
        virtual ~Empleado() {}          // destructor virtual
};

class Vendedor : public Empleado {
public:
        Vendedor( String nombre ) { /* ... */ }
        virtual void f() { /* ... */ }
        // …
        ~Vendedor() {                   // destructor virtual
                delete nombre;
        }
};

Empleado* pVendedorZonaCosta = new Vendedor;
// …
delete pVendedorZonaCosta; // OK: llama a Vendedor::~Vendedor
                           // que a su vez llama a Empleado::~Empleado
                           // siguiendo el esquema ordinal del lenguaje
```

¡Un momento! Todo esto es perfecto, pero ¿no había quedado establecido que las funciones virtuales deberían compartir exactamente la misma signatura? ¿Y no cambia, por definición, el nombre del destructor en cada clase? Naturalmente, pero este caso concreto es una necesaria excepción a tan rígida regla.

En realidad el problema con los destructores se nos puede presentar en todas las jerarquías de clases, por lo que una buena regla sería: *si de una clase se derivan, o pueden llegar a derivarse, otras clases, debemos declarar su destructor virtual*.

Hay un caso, sin embargo, en el que los destructores virtuales no nos serán de mucha utilidad: no podemos destruir un array de objetos de una clase derivada mediante un puntero a su clase base, pues el compilador no tiene forma de conocer el tamaño de los objetos de la clase derivada, necesario para calcular la situación secuencial de los punteros *this* pasados al destructor. Pero, bueno, no todo el monte es orégano.

RESOLUCIÓN ESTÁTICA DE FUNCIONES VIRTUALES

Ya hemos visto cómo convertir funciones miembros ordinarias en virtuales. Tales funciones conservan, no obstante, las características poseídas por cualquier función miembro: esto es, pueden ser invocadas directamente, sin que entre en juego el mecanismo virtual. De hecho, en algunas ocasiones la llamada a una función virtual necesariamente se resuelve en tiempo de compilación. Veamos un ejemplo con las distintas casuísticas:

```
class Base {
public:
        virtual void v() {
                cout << "función Base::v()";
```

```
        }
        Base() {}
        Base( int sinImportancia ) {
                v();                    // resolución estática siempre
        }
        Base( Base* punteroABase ) {
                punteroABase->v();
        }
};

class Derivada : public Base {
public:
        virtual void v() {      //redundante e inelegante,
                                                // aunque correcto (¡a evitar!)
                cout << "función Derivada::v()";
        }
        Derivada() : Base() {}
        Derivada( int n ) : Base( n ) {}
};

Derivada* punteroADerivada = new Derivada;
Base* punteroABase = new Derivada;
// la siguiente línea imprime Derivada::v()
//  por resolución virtual
Base objetoBase( punteroADerivada );
// la siguiente línea imprime Base::v()
//  por resolución estática
Derivada objetoDerivada( 0 );
punteroABase->Base::v();    // estática: imprime Base::v()
objetoBase.v();                         // estática: imprime Base::v()
( *punteroABase).v();                   // virtual: imprime Derivada::v()
```

En esencia, la resolución en tiempo de compilación de las funciones virtuales se reduce a los siguiente casos:
- si una función virtual se accede a través de un objeto de una clase dada, se ejecutará el cuerpo de la función correspondiente a dicha clase. Nótese en el código anterior la diferencia entre el acceso a través de un objeto y el acceso a través de un puntero desreferenciado, pues este último si usa del mecanismo virtual.
- si el acceso se produce a través de una referencia o un puntero, pero se usa el operador de resolución de ámbito (::).
- si la invocación se produce en el cuerpo de un constructor, llamado por un objeto de una clase derivada. Esto se debe a que, debido al orden de inicialización de clases, cuando se construye un objeto de una clase derivada primero se construye la porción correspondiente a su clase base: o sea, cuando se está ejecutando el constructor de la clase base todavía no se ha construido la porción del objeto correspondiente a la clase derivada, y al no existir esta porción no puede llamarse, evidentemente, a función alguna de la misma. Esto no quiere decir que cualquier función virtual en el cuerpo de un constructor se resuelva estáticamente. De hecho en el ejemplo anterior se expone un caso en que no es así, y ello se debe a que el puntero a la clase derivada, que se pasa como argumento al constructor de la clase base,

apunta a un objeto ya construido, con lo que no se da la situación anteriormente expuesta.

El lector, si acaso, podría aquí inquirir: ¿y si accedemos a una función virtual "directamente" desde dentro de una función miembro?. Algo así como

```
void Derivada::unaFuncionMiembroCualquiera()
{
        Base::funcionVirtual();
}
```

¿La respuesta? Bueno, habría que recordarle al lector que, en realidad, tal función es accedida a través del puntero implícito *this* y usando el operador de resolución de ámbito, por lo que este caso se reduce al segundo expuesto. Sin novedad, pues. Naturalmente, si suprimiéramos la porción *Base::* en el código anterior, la resolución volvería a ser dinámica (o virtual): se trataría, en definitiva, de una función virtual accedida a través de un puntero (*this*).

FUNCIONES VIRTUALES PURAS Y CLASES BASES ABSTRACTAS

Como el lector habrá podido apreciar, las funciones virtuales permiten aislar, en la clase base de una jerarquía, el interfaz genérico común a las clases derivadas de ésta. Volviendo a nuestro ejemplo primero y acercándonos más a la vida real, la clase base de nuestra jerarquía podría aparecer más o menos así:

```
class Empleado {
public:
        Empleado();
        Empleado( char* nombre );
        virtual int nivelDeResponsabilidad();
        virtual ~Empleado();
        // ...
};
```

Lo que se pretende aquí es que cada objeto de la jerarquía responda con su nivel propio de responsabilidad, y de ahí la función virtual (nótese también el destructor virtual, según lo antes comentado). Pero vayamos un poco más alla: al declarar tal función como virtual, lo que estamos haciendo también es proporcionar una implementación por defecto para las clases derivadas pues, como ya sabemos, si en una clase derivada no se redefine la función virtual, una llamada a la misma producirá la ejecución de la correspondiente a la clase base. Pero, ¿es esto lo que deseamos? Es decir, ¿existe un nivel de responsabilidad por defecto? Parece que no, que éste debería ser expresamente codificado para cada clase, pero lo cierto es que si un desarrollador descuidado añade una clase a la escala derivativa y no redefine tal función virtual, se heredará la implementación de la clase base, de manera que el resultado podría ser desastroso. Pensemos, por otro lado, en

lo que hemos querido significar con la clase *Empleado*: esto es, por ir al grano, ¿tiene sentido la siguiente expresión?

```
Empleado unEmpleado;
```

Más bien parece que no, pues cabría preguntarse: ¿un empleado de qué tipo? ¡Nadie es un simple empleado, sin cualificación absoluta! Quizá lo más apropiado sería impedir que tal clase se instanciara. Podríamos, por ejemplo, declarar sus constructores en la sección protegida, de manera que pudieran ser usados por sus clases derivadas, pero no pudieran usarse directamente por los clientes de la clase para construir objetos. De esta forma, sin embargo, podríamos vulnerar tal esquema en las clases derivadas para "exportar" al programa objetos del tipo prohibido.

¿Podemos, de alguna forma, solucionar los dos problemas expuestos? Sí, con la ayuda de las *funciones virtuales puras*. Veamos cómo una función virtual se convierte en pura:

```
class Empleado {
public:
        virtual int nivelDeResponsabilidad() = 0;
        // …
};
```

Al declarar, mediante la sintaxis expuesta, una función como virtual pura en una clase, se producen los siguientes efectos:
- No se podrán instanciar objetos de tal clase. Si ahora intentáramos la línea en la que se construía un objeto de tipo *Empleado*, obtendríamos un error en tiempo de compilación. Así, por el simple hecho de contener al menos una función virtual pura, una clase se convierte en una *clase base abstracta* o ABC (Abstract Base Class).
- Si en una clase derivada no se redefine la función virtual, se heredará la característica de "puridad" de ésta, convirtiéndose tal clase derivada en *abstracta*. De esta forma nos aseguramos que todas las clases de las que se puedan instanciar objetos redefinan, por fuerza, la función virtual de la clase base.

Cabría inquirir aquí, dada la peculiar sintaxis vista, si las funciones virtuales puras están dispensadas de ofrecer una definición. Bueno, en efecto lo están, dado que, como he explicado, cada clase con posibles instanciaciones proveerá su propio definición. Podemos, sin embargo, dotar de definición a nuestra función en la clase base, de la misma forma que lo haríamos con una función miembro ordinaria:

```
int Empleado:: nivelDeResponsabilidad()
{
        return 17;
}
```

Esta definición no sera heredada automáticamente, empero, por las clases derivadas, aunque sí podrá ser accedida por éstas mediante el operdor de resolución de ámbito:

```
class Secretario {
public:
        int nivelDeResponsabilidad()
        {       // redefinición de la función virtual pura
                return 13;
        }
        // …
};

class Administrativo {
public:
        int nivelDeResponsabilidad()
        {       // uso de la definición de la función virtual
                //  pura provista en la clase Base
                Empleado::nivelDeResponsabilidad();
        };
};
```

También podría accederse tal definición, usando del operador ::, a través de un objeto de clases derivadas, como por ejemplo:

```
Secretario secretarioDireccion;
secretarioDireccion.Empleado::nivelDeResponsabilidad();          //resolución estática
```

CONSTRUCTORES VIRTUALES

Antes hemos visto que los destructores pueden ser declarados virtuales, y es muy posible que el lector haya notado que nada a este respecto se dice de los constructores. Veamos primero qué quiere expresar la idea de un "constructor virtual". En OOP, y por ende en C++, debemos acostumbrarnos a que los objetos sean autosuficientes: si mandamos un mensaje, se ejecutará el método apropiado de la clase apropiada al objeto. Las funciones virtuales proveen buena parte de este "sabor polimórfico", estableciendo procedimientos efectivos de identificación, en tiempo de ejecución, de las funciones apropiadas a los mensajes lanzados en una jerarquía de clases. Los destructores virtuales permiten, por otro lado, despreocuparnos de codificaciones explícitas de desinicialización de objetos. En perfecta secuencia lógica, las funciones de carácter constructor y esencia polimórfica deberían liberarnos de códificar expresamente el tipo de objeto a construir. Es decir, dado un objeto a ser construido en una determinada porción de programa, un constructor virtual de una jerarquía de clases permitiría aplicar la función constructora apropiada al objeto en cada caso a partir de una codificación general. Es como si dijéramos: quien más sabe de un objeto es el objeto en sí, así que ... ¡constrúyete! Bien, esto es francamente interesante y suena casi a mágico, pero volvamos a la Tierra: es curioso pensar que un lenguaje con un muy fuerte chequeo de tipos, cual es C++, permita la creación de una codificación con una absoluta relajación de tal chequeo (piénsese que el tipo de los objetos sería determinado en tiempo de ejecución). Las posibilidades, de cualquier forma, son indudablemente atractivas. Pero vayamos al grano: C++ NO admite "constructores virtuales" (y aquí la comunidad Smalltalk podría aumentar el tono y denostar, como es habitual, la muy cacareada hibridez de C++ como OOP). Seamos más precisos: el lenguaje C++ en sí no permite que los constructores puedan ser virtuales, pero es fácil construir un esquema que pueda simular tal efecto. Bueno, puntualizando, es Stroustrup quien dice que es fácil. Básicamente la técnica más utilizada consiste en diseñar una función (o modificar el cuerpo del constructor en la clase base) que permita discernir el tipo de argumento para aplicar uno u otro constructor y devolver un objeto apropiado. A un programador tradicional esta técnica le sugeriría una estructura de tipo *switch*, aunque en C++ lo suyo es resolverlo mediante funciones virtuales. No detallaré ahora, con todo, tales mecanismos, que el lector podrá encontrar en el mismo Stroustrup y mayormente en Coplien.

11

CONCLUSIÓN

Bien, hasta aquí hemos revisado distintos aspectos del lenguaje C++ imbricándolos con la programación orientada-a-objetos, hasta llegar a lo que parece su núcleo operativo: las funciones virtuales. Y es precisamente aquí donde hemos de terminar esta introducción. ¿Por qué? Bien, precisamente debido a ese carácter deliberadamente introductorio del texto. A partir de este punto habría que considerar, al menos, los siguientes temas:

- herencia múltiple
- derivación virtual
- plantillas ("templates")
- manejo de excepciones

Pero esto supone que, a fin de sacar partido a lo expuesto, deban explicitarse cuestiones de estilo y ejemplos de limpia codificación, y no una mera exposición de la sintaxis y características del lenguaje (¿cómo explicar, sino, por ejemplo, la relajación en los tipos de retorno de las funciones virtuales en jerarquías de derivación establecida en el cuasi-estándar ANSI, y que ya se empieza a adoptar por compiladores comerciales como Borland C++ 4.0? ¿o la característica de resolución estática de los parámetros por defecto en funciones virtuales con resolución dinámica?). Naturalmente tal desarrollo habría de llevar muchísimas más páginas de las posibles en esta aproximación al lenguaje y a la programación orientada-a-objetos, por lo que habrá de quedar pospuesto, temporalmente, hasta la publicación de una nueva obra quizás denominada "Manual de estilo de C++". O quizás no. Bueno, dejen que este autor repose un poco y ya veremos.

A-1

DE LAS LIBRERÍAS Y OTROS ÚTILES

En el presente anexo hablaré, tras haber revisado sucintamente en el libro la teoría del lenguaje, de los útiles prácticos que habrían de permitir la iniciación en la programación real aplicando lo ya explicado. Pero, ¿qué enfoque tomar? Esto es, ¿debo presentar una lista comparativa, al estilo de los frecuentes artículos comerciales al uso, en que se revisan distintos productos comerciales y se les puntúa en razón de curiosos criterios, pretendidamente objetivos? Bueno, la verdad es que creo que no es esto lo que necesita el principiante en C++. Así, lo que yo simplemente voy a exponer es mi particular y subjetivísima visión de cómo debería abordarse el uso de tales productos comerciales. De esta manera todos los útiles expuestos en adelante serán revisados en razón de su conveniencia práctica para iniciar al lector en el nuevo paradigma, y no por otras cualidades más técnicas, propias de un segundo o tercer estadios en el aprendizaje del lenguaje. Me explico: en filosofía, por ejemplo, ante las obras "El Ser y el Tiempo", de Heidegger, y "Diferencia y Repetición", de Deleuze, yo recomendaría para una etapa introductoria la última, obviando, en razón del criterio elegido, las cualidades históricas y básicas de la primera obra. No recetaré, pues, una bolsa de librerías de parecido cometido, sino únicamente la que me merezca un interés diferencial. Desde luego que esta la larga perorata tiene un único fin: evidenciar que no hay recetas magistrales, y menos en este campo. Manos a la obra.

LIBRERÍAS DE CLASES

Una de las primeras ideas que se le intenta imbuir al principiante en C++ es la siguiente: "No hay que reinventar la rueda". Es prácticamente seguro que mucho de lo que se intenta implementar ya haya sido codificado por otros, de manera que no se tenga que partir cada vez de la nada. Lo más parecido a esto son las librerías de funciones al estilo tradicional. Aquí, sin embargo, operamos con objetos y, en C++, también con clases, por lo que las librerías de C++ están compuestas de clases: he ahí la primera diferencia. La segunda es significativa: aun cuando C++, mediante la derivación de clases, asegura la reutilización del código sin disponer del fuente, lo cierto es que, precisamente para favorecer la confianza del usuario, la disposición de éste, bien gratuitamente bien mediante un pequeño pago adicional, se ha convertido es una costumbre establecida. Naturalmente esto tiene su razón: aparte de las leyes que protegen la propiedad intelectual, ¿quién habría de molestarse en copiar y modificar unas clases dadas cuando, en un brevísimo lapso de tiempo, éstas serán mejoradas por la empresa que las diseñó, dedicada fundamentalmente a este proceso, ofreciendo su actualización por un módico precio? La entrega del código fuente intenta eliminar, por otra parte, la lógica desconfianza inicial del usuario de la librería de clases: el estudio de tal código asegura la posible continuidad de su desarrollo en caso que la empresa que lo creó desaparezca; se afianza, a la vez, el soporte en el que posiblemente habrá de asentarse la arquitectura software de una empresa.

Dicho esto, subyace la cuestión tal y como la plantearía el sufrido lector: "Vale, vale. Pero, ¿qué librerías debo utilizar? Bueno, en primer lugar hay que decir que a ningún programador serio de C++ (a no ser que se dedique a la confección de librerías de clases comerciales) se le ocurre implementar arboles binarios, clases "string", tablas de hash, vectores, etc., para un trabajo serio. La práctica totalidad de estas estructuras algebráicas está disponible en distintas librerías, a un precio muy ajustado y con una calidad difícil de superar. No hay que perder de vista el hecho que las clases nos permiten, mediante el mecanismo de herencia, refinar o modificar selectivamente las clases en la librería usada, pues no estamos hablando de funciones, como no me canso nunca de repetir.

Bien, existen dos tipos básicos de librerías: por un lado están las que parten de una única clase (librerías cósmicas), normalmente denominada "Object", y que suelen constituirse en los marcos de aplicación que veremos un poco más adelante; por otro lado están las librerías basadas en distintas clases. En este apartado nos centraremos en librerías del último tipo.

Antes de continuar, no debemos obviar lo evidente: la mayoría de compiladores comerciales (lisez Borland C++ 3.1, Visual C/++, Zortech C++ 3.0, CA C++, Liant C++, etc.) vienen acompañados por distintas librerías y marcos de aplicación que intentan cubrir los distintos aspectos susceptibles de aparecer en el desarrollo de una aplicación. Así, por ejemplo,

Borland ofrece los entornos TurboVision, ObjectWindows, además de una librería de contenedores tipo Smalltalk, y una versión adicional de la misma usando plantillas; mientras que Microsoft ofrece lo que denomina MFC's (Microsoft Foundation Classes), para el desarrollo práctico de aplicaciones Windows. No voy a entrar en detalles sobre estas librerías, suficientemente documentadas en sus compiladores respectivos, pero sí voy a enumerar algunas apreciaciones al respecto:
en primer lugar, el hecho de que una empresa construya un buen compilador no implica que la calidad de las librerías que distribuye sea pareja. Ni siquiera quiere decir que serán las que mejor se ajustan al compilador en concreto, pues suele ocurrir que otras librerías de firmas independientes aprovechen mejor los recursos ofrecidos; por otro lado, hay que notar que tales librerías suelen ofrecer (con alguna excepción, como "CommonView" de CA) bases para el desarrollo no-portable de aplicaciones, pues atienden demasiado, en aras de una frecuentemente no bien entendida eficiencia, a manejar los API's de los entornos para los que están especificamente diseñadas. No es esto lo que un programador de C++ suele esperar. No, al menos, al principio: ¿por qué -podría pensar el lector- debo imbuirme de conocimiento de MS-Windows para una aplicación, y más tarde perder un tiempo precioso estudiando, por ejemplo, los API's de OS/2 para poder migrar la anterior aplicación a este sistema? ¿He de inventar la rueda cada vez que cambie de sistema o entorno gráfico? ¡Naturalmente que no! No parece lógico que algo teóricamente tan trillado como, verbigracia, el manejo de ventanas en diferentes entornos, deba ser específicamente manejado por el desarrollador. No estoy diciendo que el enfoque del tipo de librerías expuesto no llegue a resultar efectivo en casos concretos, muchos o pocos, pero sí que difícilmente puede resultar didáctico para el principiante. Por esta razón, obviaré su comentario.

Tools.h 5.1, de Rogue Wave Associates.

Nos encontramos ante una librería inteligente, portable y digna de estudio. No he dicho, sin embargo, que sea la más eficiente (piénsese que, por ejemplo, existe una extensa librería denominada M++ para el cálculo matricial en C++). La presente versión incluye el manejo de plantillas (templates). La extensa lista de clases, que comienzan por 'RW' (a excepción de las genéricas, que comienzan por 'G'), incluye algunas como "RWBench" (automatiza el proceso de testear -benchmarking- una porción de código), "RWTimer" (para medir lapsos de tiempo real), "RWBtreeOnDisk" (para la gestión de árboles binarios en disco"), "RWCString" y "RWCSubString" (la versión Rogue Wave de la típica clase "string"), "RWTStack<T,C>" (plantilla de una pila de valores, donde 'T' representa el tipo de objetos en la pila, mientras que 'C' se refiere a la implementación de la pila: vector ordenado, doble lista enlazada, etc.), y otras muchas más. Veamos cómo se le puede sacar partido a esta librería.

Imaginemos que una empresa, Caos Informático S.L. (en abreviatura C.I.), necesita manejar objetos de tipo genérico "string". Lo más inmediato sería usar de forma directa la clase "RWCString", pero esto plantea sus propios problemas: si Rogue Wave cambia el nombre de sus clases en futuras versiones (como ha ocurrido con la nombrada, que antes se denominaba "RWString"), o si deseamos cambiar de librería, nos encontraremos que nuestro código cliente de tales clases deberá ser modificado. Pero esto no es lógico. ¿Cuál es, pues, la solución? Sencilla: debemos crear una clase "nuestra" que sirva de interfaz entre nuestras aplicaciones y las librerías elegidas:

```
class CIString : private RWCString {
public:
        // constructores
        // aquí se explicita el constructor
        //  adecuado de la clase base
        CIString() : RWCString() {}
        CIString( const char* cs ) : RWCString( cs ) {}
        // ...
        // seguidamente se hacen accesibles desde objetos de
        // nuestra clase CIString diversas funciones de RWCString
        RWCString::operator==;
        RWCString::hash;
        // ...
        // aquí se cambia el interfaz de alfunas
        //  funciones de RWCString
        void cambiaAMinusculas()
        {
                RWCString::toLower();
        }
        // ...
};
```

La verdad es que la clase "RWCString" consta de un número elevado de métodos propios (¡sesenta y cinco!), incluyendo indexación, subcadenas, etc. Hemos creado en un momento, pues, una clase para el manejo de cadenas totalmente pulida, y además, por la derivación privada, hemos establecido una barrera que impide el acceso al código de la librería. Pero, un momento: después de una larga serie de capítulos de introducción a C++, el avieso lector podría plantear: ¿y si, en lugar de derivación pública, planteamos una derivación protegida? O también: ¿y si trocamos la derivación por "layering", incluyendo un objeto de tipo "RWCString" en la sección privada de nuestra clase "CIString"? Pues que yo sólo podría decir: ¡Bravo, amigo lector! ¡Se nota que ya empieza a pensar antes de codificar! Veamos las dos cuestiones: en cuanto a la primera, la derivación protegida significa que el interfaz de la clase "RWCString" y de sus correspondientes clases bases será accesible en el protocolo de las clases derivadas de nuestra clase "CIString", de forma que otros componentes del equipo de desarrollo podrían establecer subinterfaces de uso de la librería, pero esto no parece, en nuestro caso, muy aconsejable; la otra posibilidad, inteligente lector, se refiere a una cuestión que Tom Cargill denomina "herencia innecesaria": ya que no vamos a favorecernos de

las ventajas que el mecanismo de herencia proporciona, ¿por qué entrar en complicaciones?. La verdad es que el planteamiento de "layering" es el más prudente en este caso, y consistiría en declarar en la sección privada de "CIString" un puntero, por ejemplo, a un objeto "RWCString", incluyendo (pues ya no contamos con los mecanismos de construcción y destrucción en derivación) código explícito para su construcción y posible destrucción en los constructores y destructor de nuestra clase. El interfaz, por otra parte, sería diseñado con total libertad, conteniendo su implementación mensajes al objeto de tipo "RWCString" creado (usando de las funciones miembros de su clase). Bien: todo esto es fácil y muy conveniente, así que, sin hacer muchos ascos, lo dejaré de elemental ejercicio para el lector, pues toda introducción que se precie ha de procurar trabajo improbo al principiante.

MARCOS DE APLICACIÓN

Un marco de aplicación (application framework) es, simplemente, una librería de clases que, más que ayudar a soportar determinados procesos e interacciones entre objetos, se establece como un "marco" que envuelve y sirve de base a la totalidad del código desarrollado. Estas librerías suelen ser de tipo cósmico y se destinan, principalmente, a la gestión de interfaces gráficos de usuario (GUI's). Precisamente debido a esto, una determinada aplicación normalmente hará uso de tan sólo un entorno de este tipo cada vez. La orientación "Smalltalk" se torna especialmente patente en estas librerías especializadas, así como algunas de las técnicas usadas en este lenguaje.

C++/Views 2.0, de Liant Software Corporation, Framingham, MA.

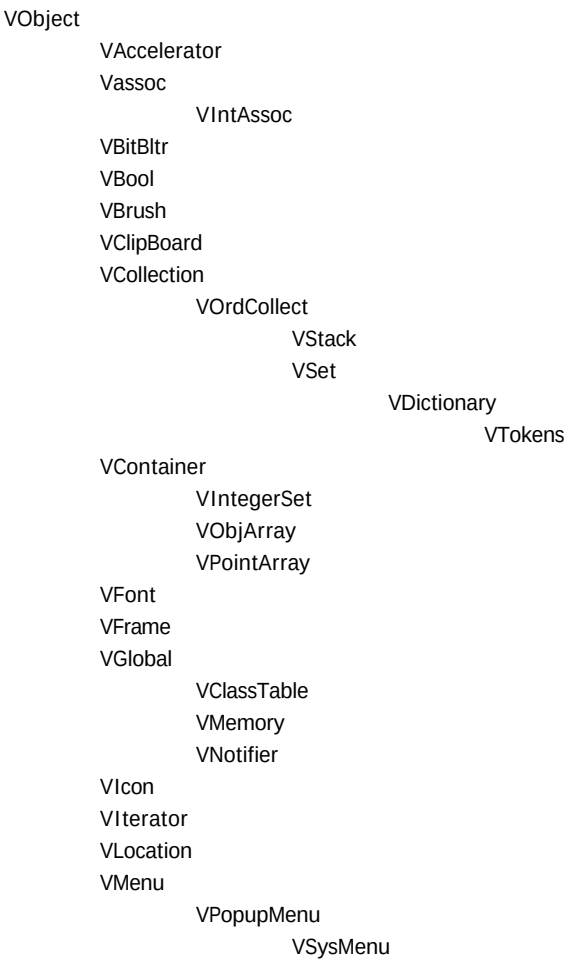
Esta es una librería que yo recomiendo fervientemente, sin contraindicaciones, como una de las mejores herramientas comerciales existentes para la iniciación a la programación orientada-a-objetos en C++. Se trata, en esencia, de un marco de aplicación para el desarrollo de programas de gestión de GUI's usando C++, pero con algunas características que le confieren un especial atractivo pedagógico:

- El mismo código fuente puede ser portado sin modificaciones, únicamente recompilando, a entornos como Microsoft Windows, OS/2, OSF/Motif, Mac, etc.
- La librería está basada en el paradigma MVC (Modelo-Vista--Controlador: Model-View-Controller), de claras facilidades para la programación en entornos gráficos.
- El producto posee una utilidad gráfica denominada "C++/Browse" que permite el examen en jerarquía derivativa de clases del código fuente, antes de compilar, con una disposición muy similar a la de Smalltalk. Esta herramienta facilita grandemente la realización de otras tareas comunes, como el "makefile", la gestión de dependencias

de ficheros de cabecera, el manejo gráfico de las jerarquías de clases, etc.
- La derivación gráfica de clases, a través de C++/Browse, es especialmente didáctica, pues refleja de forma evidente un método de reutilización de código distinto al de las librerías tradicionales.
- Las herramientas que acompañan al paquete son especialmente prácticas: una de ellas permite, por ejemplo, convertir los ficheros script de tipo dialog, generados como DLG por el SDK de Windows o una herramienta similar, en ficheros de cabecera C++ enlazadas con la librería de clases, acercándose, de algún modo, a la programación visual portable. Otra utilidad facilita la autodocumentación de las clases generadas en la aplicación.
- La documentación del producto, salvados algunos inexplicables errores, es bastante instructiva y completa.

Como quiera que parece que es un producto de este tipo el que más pudiera interesar, por la facilidad que aporta en el desarrollo de aplicaciones Windows, a un lector de RMP, me extenderé suficientemente detallándolo.

C++/Views, en su versión 2.0 para MS-Windows, está compuesto por las siguientes clases:



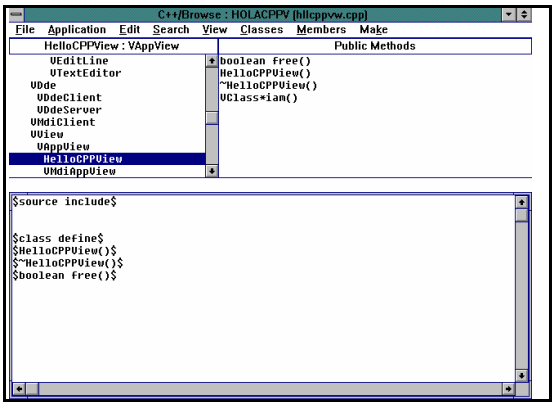
```
VMenuItem
VMouseCursor
VPen
VPort
VRegion
        VPolygon
        VRectangle
                VEllipse
                VRoundRect
VString
        Vstream
                VFile
                        VArchiver
                VSerial
                VToFromStream
                        VTagStream
                                VTokenStream
VTimer
VDisplay
        VBitMap
        VPrinter
        VWindow
                VControl
                        VButton
                                VCheckBox
                                        VRadioButton
                                        VTriState
                                VPushButton
                                        VIconButton
                        VGroup
                                VButtonGroup
                                        VInclusiveGroup
                                                VExclusiveGroup
                        VListBox
                                VComboBox
                                VMultiListBox
                        VScrollBar
                        VTextBox
                                VEditBox
                                        VEditLine
                                        VTextEditor
                VDde
                        VDdeClient
                        VDdeServer
                VMdiView
                VView
                        VAppView
                                VMdiAppView
                        VControlView
                        VMdiView
                        VPopupWindow
                                VDialog
                                        VAbout
                                        VAddRemoveList
```

VFileSelect
VInput
VListSelect
VMultiSelect
VReport
VYesNo

VEvent
VClass
VRatio
VWinInfo

Como puede inmediatamente apreciarse, existen clases predefinidas para la mayoría de los objetos que pueden aparecer en una típica aplicación gráfica, tales como diálogos SiNo (VYesNo), cajas para introducción de líneas (VInput), gestión de comunicaciones (VSerial), manejo de menús (VMenu), etc. Perfecto, pero ¿cómo se puede hacer uso de estas clases? ¡Mediante C++/Browse! Efectivamente, con esta herramienta tendríamos en la pantalla tres ventanas: una lista de clases, dispuestas gráficamente en jerarquía, una lista de métodos, ordenados alfabéticamente (y de aquí la conveniencia de codificar el nombre de nuestras funciones en inglés), y un editor de texto. Lo que se evidencia también enseguida es que los nombres de todas las clases comienzan con 'V': esto es lógico y recomendable, pues la anterior versión del producto mantenía clases como "String" a secas, lo que ocasionaba problemas al usar conjuntamente C++/Views con otras librerías de clases de iguales ansias genericistas.

Vamos a construir, pues, como ejemplificación de las facilidades de este entorno de desarrollo, una ventana Windows con el texto "Hola C++" en ella, como mandan los cánones de introducción a lenguajes de programación.



C++/Views: Creación de una nueva aplicación

Bien, ¿qué queremos? Pues, en primer lugar, construir una ventana general de aplicación. Si revisamos la jerarquía de clases encontramos que tales ventanas están representadas en la clase "VAppView", de manera que para particularizar esta clase lo que hacemos es derivar de la misma una nueva clase. Pero, ¿cómo derivamos? Muy fácil: primero seleccionamos en la Lista de Clases la superclase, o clase de la que queremos heredar (en este caso, "VAppView"); seguidamente abrimos el menú llamado "Classes" y seleccionamos "Add Subclass": de inmediato aparece una caja de opciones preguntando si la derivación es pública y/o virtual; sigue una caja de diálogo inquiriendo el nombre de la nueva clase, que podría ser "HelloCPPView";

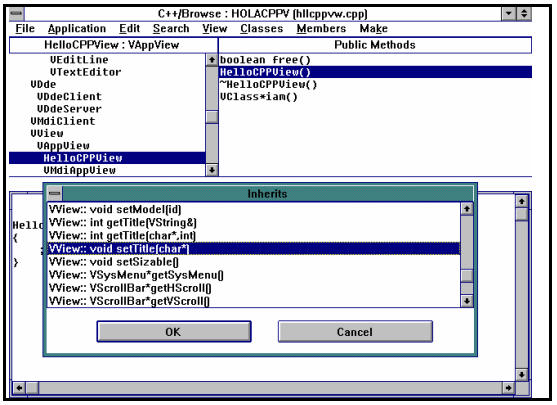
después el sistema nos pregunta el nombre del fichero en que se guardará tal clase (una de las pocas restricciones de C++/Views es que C++/Browse sólo admite gestionar directamente ficheros que contengan una sola clase), proporcionando un nombre por defecto ("hllcppvw") a través de un método simplísimo de eliminación ordenada de vocales. En seguida notaremos que la nueva clase se ha incorporado gráficamente en la Lista de Clases como subclase de "AppView", a la vez que en la Lista de Métodos han aparecido automáticamente tres funciones:

```
HelloCPPView()
~HelloCPPView()
boolean free()
```

¿Qué ha pasado? Pues nada: simplemente que el sistema, cada vez que se crea una nueva clase, provee, en primer lugar unas ciertas funciones por defecto, como son el constructor sin argumentos y el destructor, que, en realidad, tienen el cuerpo vacío, por lo que parecen ¡no hacer nada!, como si fueran plantillas a ser rellenadas por el desarrollador, aunque, como en seguida veremos, esto no es así. El sistema también provee una función con el siguiente código:

```
boolean HelloCPPView::free()
{
        delete this;
        return( TRUE );
}
```

tratándose, en el fondo, de una función virtual de cometido similar al del operador "delete" de C++. Un poco después veremos qué significa el valor de retorno de esta función.



C++/Views: Lista de funciones heredadas

Bueno, ya hemos añadido nuestra clase. Y ahora, ¿qué? ¿qué hacemos para generar la ventana? Lo cierto es que no tenemos nada que hacer, pues la ventana de nuestra aplicación ¡¡ya está codificada!! Y aquí el desprevenido lector bien podría exclamar "¿Hay algo que se me ha escapado?". Bueno, quizás sí. Repasemos brevemente algo de lo supuestamente aprendido en este libro. La clase "HelloCPPView" deriva públicamente de "VAppView", y esta última al construirse crea una ventana de aplicación. Si recordamos el orden de aplicación de constructores en jerarquías de derivación, constataremos que se ejecutan primero los de las clases bases hasta llegar a la más derivada: de esta forma para construir un objeto de nuestra

clase "HelloCPPView" se construirá antes, resumiendo, la parte correspondiente a la clase "VAppiew" (o sea, una ventana de aplicación) en aplicación de su constructor por defecto, y por último se ejecutará el cuerpo del constructor de nuestra clase (en este caso, vacío). Vemos con qué facilidad, debido a los mecanismos estudiados, hemos construido una ventana, con sus botones de maximizar y minimizar, y su menú de sistema, barra de título y marzo redimensionable. Pero sigamos, porque la verdad es que no deseamos una ventana estándar. Primero deseamos que la ventana disponga de un título apropiado, tal como "Ventana de Saludo" y, segundo, en ella debe aparecer el texto "Hola C++". Bueno, la verdad es que esto es muy fácil. El hecho que la ventana deba constar de título lo asociamos de manera evidente a la construcción de la misma, por lo que para lograr tal fin únicamente deberemos añadir el código apropiado al constructor de nuestra nueva clase. Así, seleccionando con el ratón el constructor por defecto en la Lista de Métodos, el cuerpo de éste (con un único ';') aparece en la Ventana del Editor (normalmente el Notepad), listo para ser modificado. Lo único que tenemos que hacer es incorporar en el constructor un método para cambiar el título de la ventana. Y, ¿de dónde sacamos tal método? Bien, es fácil. Tenemos dos opciones: si suponemos un suficiente conocimiento de la librería de clases del producto, inmediatamente localizaremos en la clase "VView" de la que "HelloCPPView" indirectamente deriva, la siguiente función:

```
void setTitle( char *s );
```

y cuyo cometido es, con cierta lógica, establecer el título de la vista a que se aplica en la cadena apuntada por 's'; otra opción es seleccionar, dentro del menú "Classes", la opción "Inherits...", lo que provoca la aparición de una ventana de diálogo conteniendo una lista con todas las funciones miembros heredadas por la clase actual (en nuestro caso, "HelloCPPView"), debiendo buscar seguidamente una función del tipo necesitado (esto es, algo así como "setWindowTitle" o "setViewTitle" o, definitivamente, "setTitle"), y encontrando el método anteriormente expuesto. Ahora lo único que tenemos que hacer es incluir tal método en el constructor de nuestra clase, de la siguiente forma:

```
HelloCPPView::HelloCPPView()
{
        setTitle( "Ventana de saludo" );
}
```

de manera que una vez construida la ventana, en razón de la aplicación de los constructores de las porciones de las clases bases de la nuestra, entrará en liza este último constructor, cambiando su título de la forma establecida.

Bueno, lo siguiente es añadir a la ventana la frase "Hola C++". Con la misma secuencia anterior, encontramos en la clase "VWindow", de la que "HelloCPPView" indirectamente deriva, la siguiente función:

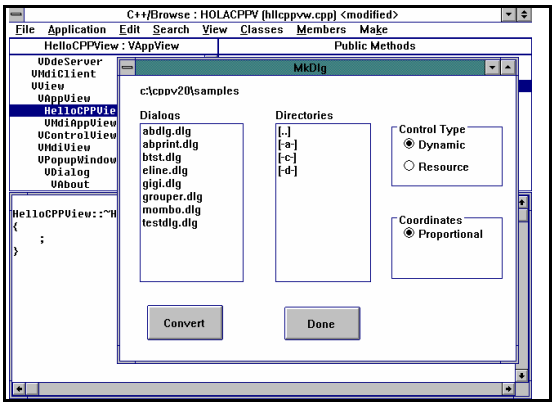
```
void wrtText( char *s, int x, int y );
```

cuyo cometido es escribir una cadena de tipo 'C' dentro del área de cliente de una ventana de tipo "VWindow" en la posición (x,y). Parece que lo tenemos claro: incluimos una llamada a esta función en nuestro anterior constructor y ... ¡voilà! ¡c'est tout!. Diantre, ¡¡no!! Recapacitemos ligeramente. Si incluimos en el constructor este método, cuando nuestra ventana se construya en ella efectivamente aparecerá la frase deseada, pero ¿qué pasará cuando esta ventana sea modificada o cubierta por otra y posteriormente descubierta? Y aquí el lector podría pensar: "Pues que la frase seguirá donde estaba, pues de eso tiene que ocuparse el sistema". Bueno, algo de razón tiene el confiado lector, pero también, en una buena medida, bastante de lógica le falta: el sistema no tiene por qué saber si deseamos que tal frase permanezca tras una modificación de la ventana; o, ni siquiera, saber si debe redimensionar el tipo de letra para adaptarlo al nuevo tamaño de la ventana. Bien, parece que antes de seguir debemos revisar el paradigma MVC en que se basa esta librería.

La arquitectura de diseño de programas MVC (Modelo-Vista-Controlador), proveniente de Smalltalk, en que se basa C++/Views, define la separación de la gestión de eventos de un programa (la capa de Control), la presentación de los datos de la aplicación al usuario (la capa de la Vista) y el modelado actual de los datos y procesos de la aplicación (la capa del Modelo). La interacción entre las distintas capas funciona como sigue: el Controlador envía mensajes a la Vista que, a su vez, ocasiona mensajes direccionados al Controlador; el Modelo, por fin, es accedido y actualizado mediante mensajes provenientes de la Vista. De esta forma se evidencia que los objetos pertenecientes a la capa Modelo son pasivos con respecto al sistema, accedidos únicamente en su relación con éste a través de los objetos de la capa Vista, que mantienen una referencia a aquéllos. El Controlador, por último, provee el interfaz entre dispositivos de entrada (teclado, ratón, etc.), actividades de alto-nivel (eventos de movimiento, arrastres del ratón, etc.) y las capas de Modelo y Vista. Veamos cómo C++/Views implementa este paradigma.

- Controlador: la gestión del control de la aplicación y de los eventos se centraliza en la clase denominada "VNotifier". Existe un único objeto de tipo "VNotifier" para cada aplicación, creado en el fichero "globals.cpp" y al que apunta el identificador "notifier".
- Vista: la presentación de una aplicación recae en la jerarquía de clases encabezada por la clase "VWindow", y se sustancia en controles ("VButton", "VListBox", VTextEditor", etc.), diálogos ("VYesNo", "VAbout", "VFileSelect", etc.), menús ("VMenu", "VSysMenu", etc.), estructuras de datos ("VStack", "VCollection", etc.) y ABC's (clases bases abstractas) extensibles por el usuario ("VWindow", "VPopupWindow", "VAppView", etc.).

- Modelo: aquí vendrían las clases específicas, implementadas por el usuario, que encapsularían los algoritmos, procesos y datos de la aplicación a desarrollar.



C++/Views: Utilidad de conversión WinMkDlg

Bien: examinemos, desde un punto de vista global, cómo opera un tal sistema. Lo que, en definitiva, se está haciendo aquí es combinar la programación por eventos con la programación orientada-a-objetos, de acuerdo con el siguiente esquema: los eventos son capturados por el objeto de tipo "VNotifier", y son trocados por éste en mensajes polimórficos (esto es, funciones virtuales) que se dirigen al objeto que actualmente retiene el "focus" de la aplicación, de tipo derivado de "VWindow" (para que pueda funcionar en C++ el mecanismo virtual). Así, la clase "VWindow" y sus derivadas implementan, al menos, una función virtual para responder a cada mensaje generado por el "notifier", causado, a su vez, por un evento. A fin de permitir operaciones con una adecuada cantidad de instancias de clases, las tablas virtuales son colocadas, si es posible, en un segmento distinto. Hay que tener muy en cuenta que un marco de aplicación no es, por ejemplo, un simple "Windows-Builder" o una herramienta lower-case (o pseudo-x-case) generadora de código para entornos gráficos. Se trata en este caso, más bien, de un entorno para dotar de productividad a la programación orientada-a-objetos en C++ y, como proclama Liant, "todo lo que pueda hacerse en C++ puede hacerse en C++/Views". La clase "VNotifier" provee también servicios a clases del tipo genérico "VWindow" para controlar secuencias de eventos y redirecciones teclado/ratón. Se trata, desde mi punto de vista, de uno de los mecanismos más elegantes para solucionar la a veces difícil combinación de programación dirigida a eventos y OOP. Y piensen en la antiguedad (Smalltalk-80) del paradigma. C++/Views proporciona al usuario, además, una herramienta utilísima denominada MKDLG en su versión DOS y WINMKDLG en su versión Windows, que básicamente consiste en un conversor de ficheros script de tipo DLG, generados por el SDK o equivalentes, a ficheros *.H y *.CPP: esto es, los cuadros de diálogo se generan de forma visual, con una herramienta apropiada, y una vez que generamos el fichero script correspondiente, C++/Views lo transforma en una clase de tipo VDialog, con su pertinente ficheros de cabecera e implementación, enlazados con la librería de clases del entorno. Se consigue así una semi-programación visual.

Volvamos, sin entreternos más, a lo práctico. Imaginemos que una ventana cubierta queda descubierta, o que se maximiza una ventana minimizada: ¿qué pasa? Pues, entre otras cosas (como la adjudicación del "focus", etc.),

el "notifier" envía el mensaje "paint()" (píntate) a la ventana apropiada, esperando que ésta responda de la forma más apropiada (mecanismo virtual). Naturalmente la clase "VWindow" define el método virtual de la siguiente forma:

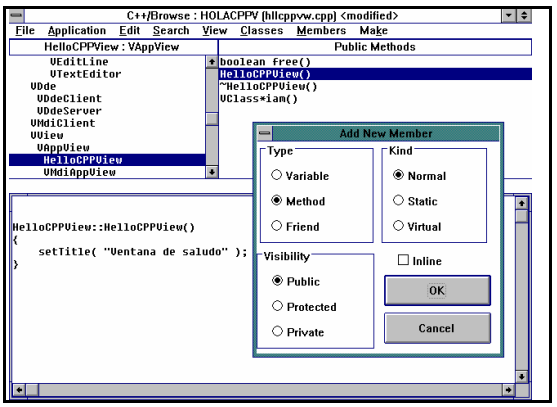
```
boolean VWindow::paint()
{
        return ( FALSE );
}
```

estableciendo la respuesta por defecto al mensaje de "píntate", si la función virtual no ha sido redefinida en la clase a que pertenece nuestra ventana concreta. ¡Vaya! Nos encontramos de nuevo con el retorno booleano. Vamos a la explicación: si el método en cuestión consume el evento, debe retornar TRUE, mientras que si no actúa sobre el evento debe devolver un valor FALSE, permitiendo así que el sistema subyacente aplique la respuesta por defecto a tal evento.

Ahora podemos volver al método "free()" en nuestra clase "HelloCPPView". En principio debemos decir que la está función está originariamente definida en la clase "VObject" con la misma codificación que en nuestra clase. En realidad vemos que se trata de un mecanismo de destrucción de objetos que, por razones de la arquitectura del entorno, debe ser reimplementado en cada clase, de manera que cuando el sistema envíe el mensaje de liberación se destruya el objeto apropiado. Ahora se comprende que el sistema codifique esta función por nosotros.

Bien, abordemos ahora el tema de la frase de saludo. Dado que el mensaje "píntate" se envía por el "notifier" a una ventana tras su creación o después de cualquier modificación, lo procedente sería establecer un método de respuesta a tal mensaje que, simplemente, dibuje nuestro mensaje de saludo en la ventana. ¿Cómo haremos esto en C++/Views? Mediante la opción de "añadir miembro" a nuestra clase. Veámoslo:

```
boolean
HelloCPPView::paint()
{
        wrtText( "Hola C++", 20, 20 );
        return ( TRUE );
}
```



C++/Views: Adición de un miembro a una clase

Mediante la última sentencia estamos diciéndole al sistema que "consumimos" el evento, por lo que no se debe aplicar acción alguna adicional. Bien, pues ya está todo: ya tenemos la aplicación. ¡Un momento, un momento! -podría intervenir aquí el infatigable lector-, ¿dónde está la

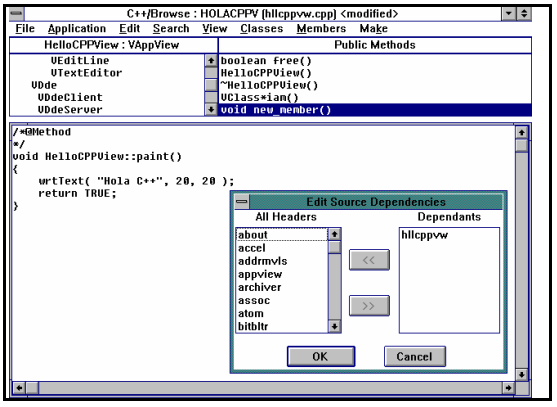
función principal?. Es cierto: cómo casi siempre el lector ha dado en el clavo. Nos hace falta tal fichero con tal función, pero lo cierto es que el sistema ya se ha ocupado de ello. En realidad C++/Browse provee una plantilla genérica para la función principal (denominada "cvmain"), que se reduce a las pocas líneas siguientes:

```
AppView *v = new AppView();
v->show();
notifier->start();
return TRUE;
```

más los correspondientes archivos de cabecera. Lo que se exige al usuario es que sustituya las menciones a "VAppView" por las de su clase específica (en nuestro caso "HelloCPPView"). Listaré, a continuación, los tres ficheros generados:

```
// ----------------------------------------------------------------
//                                          HOLACPP.CPP
// ----------------------------------------------------------------

#include "notifier.h"
#include "hllcppvw.h"

char *CTWindow = __FILE__;

#ifdef TURBO
        int i;
#endif

int cvmain(int ac, char **av)
{
        HelloCPPVView *v = new HelloCPPVView();
        v->show();
        notifier->start();
        delete v;
        return(TRUE);
}


// ----------------------------------------------------------------
//                                          HLLCPPVW.H
// ----------------------------------------------------------------

#ifndef hllcppvw_h
#define hllcppvw_h
#include "appview.h"

CLASS HelloCPPView : public VAppView {
public:
        VClass *iam();
        boolean free();
        HelloCPPView();
        ~HelloCPPView();
```

```
        boolean paint();
};

extern VClass *HelloCPPViewCls;
#endif /* hllcppvw_h */


// ----------------------------------------------------------
//                                          HLLCPPVW.CPP
// ----------------------------------------------------------

#include "hllcppvw.h"

defineClass(HelloCPPView,VAppView)

HelloCPPView::HelloCPPView()
{
        setTitle( "Ventana de Saludo" );
}

HelloCPPView::~HelloCPPView()
{
        ;
}

boolean HelloCPPView::free()
{
        delete this;
        return( TRUE );
}

boolean
HelloCPPView::paint()
{
        wrtText(
        "Hola C++", 20, 20 );
        return( TRUE );
}
```

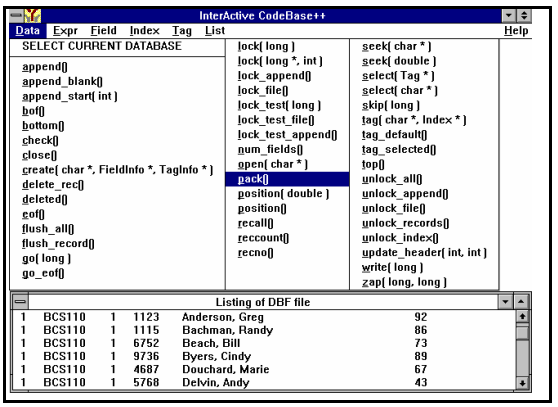


C++/Views: Editor de dependencias

Y esto es todo. Pero lo mejor es que el mismo código, recompilado con el entorno C++/Views apropiado, sirve para MS-Windows, OS/2, OSF/Motif, etc. Es interesante notar, también, que C++/Views genera automáticamente un fichero make para esta aplicación. Nótese también que cada vez que se añade una función miembro a una clase usando C++/Browse, este incluye el prototipo de tal función en el archivo de cabecera de la clase apropiada. El entorno, también, controla las dependencias de las clases con respecto a sus ficheros de cabecera. En fin, un encanto de herramienta.

Bien, después de considerar todo lo anterior, lo cierto es que en algún momento en el desarrollo de una aplicación habrá que tratar con el archivo y recuperación de los datos, o aún mejor de los objetos. Mostraré dos distintas posibilidades: una híbrida basada en el modelo relacional y otra sustanciada como OODBS (Object-Oriented DataBase System).

Codebase++, de Sequiter Software Inc., Alberta, Canada.



Codebase++: Selección interactiva funciones miembros

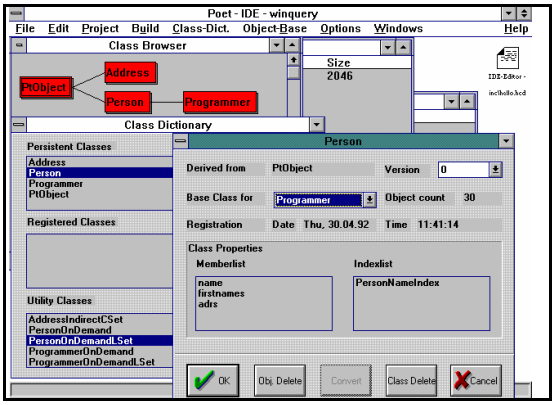
No es ésta una librería que yo aconseje sin antes despachar, como en el más común de los medicamentos, una advertencia sustanciada en la frase: "Úsese si no hay más remedio". Lo que, en definitiva, intento explicar es que la presente librería es, más que una creación originaria en C++, una remodelación de una exitosa librería de la misma empresa denominada Codebase, codificada en C y destinada al manejo de ficheros de formato DBF. El estudio detallado de las clases de esta librería evidencia que simplemente se han aprovechado las mejoras del lenguaje C++ con respecto a C, pero que no se ha aplicado la esencia constituyente del paradigma de orientación-a-objetos. De hecho, Sequiter pone más énfasis en sus productos en C que en los de C++. Bien, vayamos al grano. Para que el lector pueda hacerse una idea, examine el siguiente programa, cuyo cometido es eliminar la marca de borrado de los registros de una base de datos relacional sustanciada en un fichero DBF:

```
#include "d4data.h"

extern unsigned _stklen = 10000;        // salva bug de Borland C++

int main( int, char** )
{
        // se crea un objeto de tipo CodeBase, que contiene
        //  los parámetros comunes a todo CodeBase++, mayormente
        //  sustanciados en el manejo de errores y flags.
```

```
        CodeBase cb;
        // se asigna un identificador
        //  e inicializan variables internas
        DataIndex miDBase( &cb );
        // abre el fichero adecuado y lo asigna a la base de datos
        miDBase.open( "FICHERO.DBF" );
        // el siguiente bucle recorre la base de datos, desde el
        //  principio (top) hasta el final (eof),
        //  de registro en registro
        for( miDBase.top(); !miDBase.eof(); miDBase.skip() )
                // quita la marca de borrado de un registro
                miDBase.recall();
        return 0;
}
```

Aquí, por supuesto, vale lo dicho con respecto a la librería de Rogue Wave: una vez decididos a usar CodeBase++, debe crearse una clase interfaz nueva (quizás CIDBFM: Caos Informático DataBase Files Manager) que encapsule los métodos de gestión necesitados: abrir, cerrar, indexar, filtrar, etc.

Poet, de BKS Software Entwicklungs GmbH, Berlin.



Poet: Browsers gráficos y textuales de clases

Esta es una herramienta más apropiada que la anterior, según el esquema conceptual preconizado por la OOP, para dotar de persistencia a los objetos de una aplicación. Se trata, en síntesis, de un preprocesador denominado PTXX que permite extender el lenguaje C++ mediante la adición de palabras clave para significar la persistencia. El desarrollador opera con un fichero de cabecera con extensión HCD que, tras pasar por el preprocesador, genera un fichero HXX en el que las palabras clave nuevas han generado relaciones de derivación, clases PTQuery, etc. El entorno, bajo Windows, cuenta con browsers de clases gráficos y textuales, que permiten la identificación de jerarquías, de clases persistentes, etc.

La sintaxis es transparente:

```
persistent class MiClase {
        // …
};
```

de tal forma que esta simple adición asegura que la nueva clase dispondrá, entre otras cosas, de un método "store()" para el archivo expreso de sus instanciaciones.

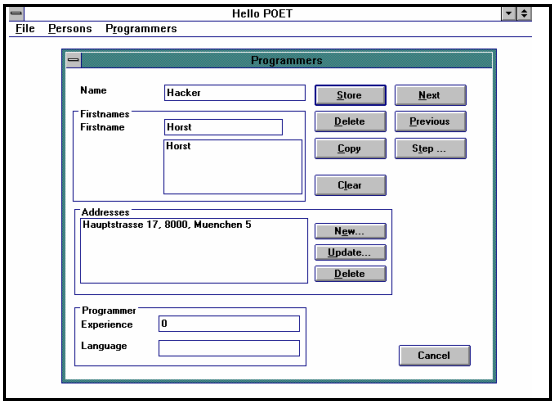
```
PtBase baseDeObjetos;
// …
// el preprocesador ha creado un nuevo constructor que toma
// por argumento un objeto PtBase
MiClase* objetoDeMiClase = new MiClase( baseDeObjetos );
objetoDeMiClase->store(); // archivo en la base de objetos
// …
delete objetoDeMiClase;
```

a la vez que se han creado, por ejemplo, clases de queries semánticamente asociadas a nuestra clase, a través de los que podemos recuperar el o los objetos deseados de una base de objetos:

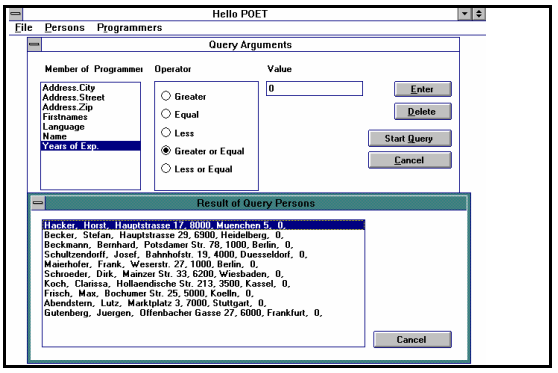
```
// PTXX genera la clase [NOMBRECLASE]AllSet, que es una
// clase contenedora de objetos de tipo MiClase
MiClaseAllSet* allMisClases =
        new MiClaseAllSet( baseDeObjetos );
MiClase* unObjeto;
// posiciona el contenedor al principio
allMisClases->Seek( 0, PTSTART );
// establece un bucle para revisar
//  todos los objetos del contenedor
while ( allMisClases->Seek( 1, PTCURRENT ) == 0 )
{
        // obtiene el objeto correspondiente
        allMisClases->Get( unObjeto );
        unObjeto->hazAlgo();
        // seguidamente se descuenta una referencia de apuntadores
        // al objeto y, si nadie lo usa, se destruye
        allMisClases->Unget( unObjeto );
```

```
}
delete allMisClases; // destruye el contenedor
```

El producto instala, por defecto, tres ejemplos básicos de uso del motor de persistencia, entre los que destaca una implementación de una "base de objetos" de "personas" y "programadores", donde estos últimos forman un subconjunto de aquéllos. La ejemplificación del interfaz de introducción y edición de los objetos es adecuada, y las facilidades provistas para las consultas, mediante "queries" guiados es suficientemente clara. Ciertamente los interfaces gráficos pudieran hacer pensar que se trata aquí, sin más, de una típica descomposición en tablas matizada por un barniz comercial de orientación a objetos. Bueno: no es así, pero esto dice mucho a favor de la capacidad estandarizadora de tales interfaces. Realmente los objetos de tipo "programador", por ser subtipos de "persona", asimilan, en la práctica, las posibilidades, matizadas por las cualificaciones de acceso, de estos últimos objetos. Se trata, en una conclusión informal, de la persistencia de la herencia, donde los objetos de tipo "programador" aprovecharían la implementación de la persistencia en los objetos "persona", dada la relación de derivación pública que une ambas clases. Este ejemplo, como los dos anteriores, es realmente instructivo sobre la facilidad que herramientas de este tipo, y aun más potentes y no basadas en esquemas extensivos del lenguajes, pueden procurar al desarrollador.

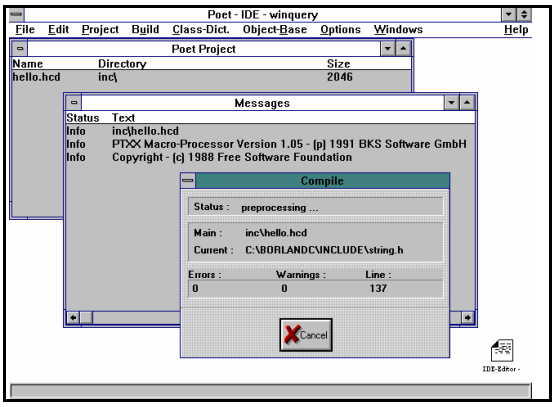


Poet: Base de objetos "personas" y "programadores"



Poet: Query sobre una base de objetos

Como fácilmente se puede apreciar, el preprocesador realiza mucho trabajo por nosotros, generando un conjunto de clases que comparten el nombre de la clase preprocesada y que servirán para interactuar con la base de objetos. Algunas de las palabras claves nuevas son: persistent, ondemand, transient, depend, etc. POET añade, además, por sus características de preprocesador, soporte para plantillas (templates) independientemente de si el compilador concreto está capacitado para mantenerlas. Se trata, al fin, de un producto muy interesante, multiplataforma y de precio muy ajustado. Un detalle más extenso de sus características, por basarse en conceptos no revisados en esta libro (OODBS's en contraposición al modelo relacional), no cabe en el presente anexo, aunque, por su interés, es merecedor, sin duda, de un capítulo diferencial.



Poet: Compilador-preprocesador de persistencia

A-2

BIBLIOGRAFÍA COMENTADA

El boom editorial generado en USA por la tecnología de objetos ha poblado las librerías de ese país de títulos en los que abundan los términos "C++" y "Orientado-a-Objetos". La confusión que tal profusión de obras puede causar en el inadvertido lector es inevitablemente grande. La elección se torna difícil y su resultado incierto. Más del 50% de los textos son, sencillamente, infames. Un 30% adicional son, simplemente, triviales. Un 15% más se limitan a repetir viejos conceptos aplicados a implementaciones particulares, del tipo "Programe en C++ con Microsoft C/C++ 7.0", etc.. Únicamente el restante 5% contiene textos realmente interesantes, pero aun así es importante evitar malgastar un tiempo precioso con obras que exponen ideas básicamente iguales. Vamos a iniciar, pues, un recorrido por los textos de posible mayor interés para el lector. Una última advertencia: la totalidad de las obras que aquí se van a detallar están en inglés, aunque esto es algo a lo que el programador de C++ debiera pronto acostumbrarse. Desafortunadamente la escasa oferta editorial a este respecto en castellano no es, de forma general, recomendable como opción para los desarrolladores serios. Prescindiendo de los mayormente impresentables manuales de uso de compiladores, las actuales obras sobre C++ originales en nuestro idioma son, en el momento de escribir este libro, a saber: una muy elemental introducción general a la OOP proporcionada por la obra "Programación Orientada-a-Objetos", cuya segunda parte ofrece un intérprete denominado IMO con el que se pueden ensayar mensajes en calidad de "toy tool"; existe, también, una elemental iniciación a los tópicos más frecuentes de la sintaxis del lenguaje C++, escrita por F.J.Ceballos, referida a una versión claramente desafasada del mismo y basada en un equivocado enfoque de diseño top-down, suficientemente denostado y superado a estas alturas; existe, al fin, una obra que sí merece ser tenida en cuenta: "Programación en C++", de Enrique y José Hernández Orallo, que, lejos de trasnochados esquemas comerciales, ofrece un pedagógico y ejemplar panorama de introducción al "mero" C++, ajustado a AT&T C++ 3.0.

LIBROS BÁSICOS SOBRE C++

Seguidamente se detallan las obras de introducción al lenguaje, así como aquéllas de referencia básica, sin las cuales no se debiera seguir avanzando en el mismo. Naturalmente he debido, pues éste es el segmento editorial más poblado, tomar alguna decisión de exclusión que a alguno puede extrañar. No he incluido, por ejemplo, la famosa obra introductoria de Dewhurst & Stark, pues no ofrece características diferenciales con respecto a la de Lippman, mientras que ésta me parece de mayor claridad pedagógica. He procurado presentar estos textos en orden de dificultad creciente.

The C++ Workbook, por Richard S. Wiener & Lewis J. Pinson, 1990, Addison-Wesley, 0-201-50930-X, 349 pág.

He de confesar que le profeso cierto cariño a este texto, sin duda el más elemental de todos los aquí comentados. Se trata de una introducción a AT&T C++ 2.0, con multitud de ejemplos, letra grande y legible, y abundantes resultados de programas. El libro se caracteriza por frecuentes inserciones de apartados "¿Qué pasaría si ...?", que intentan resolverle al lector las dudas más habituales en el primer acercamiento al nuevo lenguaje. Teniendo en cuenta la proliferación actual de desenfocadas introducciones al lenguaje C++, repletas de dudosas intenciones y de tontería, esta obra es un respiro que se puede completar en escasas horas. No sustituye, por supuesto, a textos de introducción como el de Lippman o el de Dewhurst & Stark, pero muy bien podría constituirse en el perfecto puente hacia éstos.

C++ Primer, 2nd Edition, por Stanley B. Lippman, 1991, Addison-Wesley, 0-201-54848-8, 614 pág.

Nos encontramos ante lo que podría ser calificado como el texto estándar (por oficial) de introducción a C++. En él se detalla la especificación AT&T C++ 3.0 (y, por tanto, plantillas y clases anidadas) de una forma extremadamente rigurosa y pulida: los escasísimos errores o carencias del texto se han debido, como es posible apreciar en comp.lang.c++, a circunstancias ajenas al autor. No se le supone al lector conocimiento previo alguno de C++ (ni siquiera una media experiencia en C), y la exposición gradual de las características del lenguaje está perfectamente modulada. O sea, se trata del libro ideal para iniciarse, sin la ayuda externa de un profesor (aunque ésta es, en general, de valiosísima consideración), en los tópicos y recovecos del lenguaje. Lippman empieza muy pronto en la obra con la codificación de la clase "Array", y el lector puede asistir al proceso de su refinamiento, mediante la aparición en el texto y acertado comentario de nuevas características que se le van añadiendo, involucrándose y comprometiéndose poco a poco en el proceso. El libro está repleto de instructivos ejercicios, de factible resolución, a la vez que de ejemplos "vivos": el lector ve, sorprendido, cómo éstos van incorporando con extrema facilidad nuevos detalles conforme avanza el relato. Todas las secciones del libro muestran en la práctica, pues, los conceptos teóricos que se desprenden del lenguaje. Se exponen, por otro lado, una buena cantidad de interesantes y bien probados trucos de programación, que el lector podrá apreciar "en su salsa". Lippman dedica también un capítulo a OOD (Diseño Orientado-a-Objetos), sustanciado, como es habitual en el resto del libro, en un ejemplo apropiado. La técnica de OOD empleada es simplísima: se trata de identificar las clases necesarias a nuestra aplicación para dotarlas seguidamente de los interfaces apropiados, y establecer las relaciones entre ellas. Se trata, pues, más que de una exposición de OOD, una indicación de la importancia que se le debe conceder al uso, expreso o no, de los conceptos de OOD a la codificación en C++. En definitiva: este libro es fundamental para el principiante, mientras que para el experto se convierte en paradigma de cómo debe escribirse un buen texto de introducción.

The C++ Programming Language, 2nd Edition, por Bjarne Stroustrup, 1991, Addison-Wesley, 0-201-53992-6, 669 pág.

Esta es la segunda edición del texto que el Dr. Stroustrup publicó en 1.986 detallando el lenguaje C++, en su calidad de creador del mismo. Se trata de un tutorial del lenguaje en el que, a diferencia del texto de Lippman, se enfatizan los aspectos claves de uso del mismo. Se asume que el lector tiene experiencia previa en programación en C, y se detalla la especificación AT&T C++ 3.0 partiendo "de cero". El estilo del texto es enormemente sintético, de manera que la cantidad de tópicos revisados es netamente superior a la del texto de Lippman. Comparada con la primera edición, la obra ha crecido también considerablemente en número de páginas. Precisamente ahora,

cuando se empieza a generalizar el uso de plantillas (templates) y se empieza a asentar la posibilidad práctica de uso del tratamiento de excepciones, las páginas de este libro sobre tales materias se constituyen en una insustituible guia referencial a la vez que conceptual, pues el conocimiento de la mera sintaxis es claramente insuficiente. Si en la primera edición se revisaba la librería que entonces se denominaba "stream", implementada por el propio Stroustrup, aquí se repasa la librería "iostream", sobre las bases desarrolladas por Jerry Schwarz (que, por otro lado, presenta continuas revisiones al comité ANSI C++). Se muestran y discuten algunos interesantes trucos y técnicas del lenguaje y el texto termina con un manual de referencia. Una sección importante del libro la constituye (120 páginas) la dedicada a OOD (Diseño Orientado-a-Objetos), en la que, usando de una extensión de la técnica de las fichas CRC (de la que el lector encontrará una sucinta descripción más adelante, en el texto de Wirfs-Brock), se introduce con acierto al lector en esta disciplina. Bien, se trata de un libro muy difícil de resumir por su buscada tersedad conceptual y de estilo, pero es indiscutible que, quizá con el soporte previo de una obra introductoria como la de Lippman, su posesión es indispensable para cualquier programador de C++. Estudiénlo, sin más condiciones.

The Annotated C++ Reference Manual, por Margaret A. Ellis & Bjarne Stroustrup, 1990, Addison-Wesley, 0-201-51459-1, 447 pág.

Esta es la biblia del C++: el texto de referencia indispensable para cualquier estudioso, interesado o programador del lenguaje. Y biblia es aquí un término usado con intencionada literalidad: acaloradas discusiones entre miembros de la comunidad C++ suelen zanjarse con citas del tipo "la sección 13.2 de ARM (el apelativo cariñoso del texto) dice: ...", que surten el efecto del mágico ensalmo que sólo puede emanar de la autoridad por todos reconocida. El texto de este libro fue tomado, en su momento, como documento base por el comité ANSI C++ para la estandarización del lenguaje, y la carencia, en estos momentos, de tal estándar ha redundado en confirmar como inamovible tabla de salvación, en el complicado mar sintáctico y conceptual de C++, a esta obra. Nos encontramos ante un texto eminentemente referencial, que no pretende enseñar al lector a usar los mecanismos del lenguaje, sino más bien a describirlos en una forma tersa y rigurosa, pero que, fundamentalmente en las notas, explicita cuestiones sobre la naturaleza de C++ que pueden llevar al lector a comprender mejor la esencia del lenguaje: por qué tal o cual característica no ha sido contemplada, o por qué tal otra ha sido implementada de esta determinada manera, o cómo podrían suplirse comportamientos no previstos en el nudo lenguaje, etc. Nótese que la primera línea del prefacio dice así: "Este libro provee una completa referencia del lenguaje para el usuario experto de C++". No se pretende que el lector estudie de golpe, del principio al final, el texto, pues tal tarea sería como la de intentar abarcar por orden alfabético una vasta enciclopedia (y esto trae reminisencias, no obstante, de aquel "autodidacto" de la nausea sartriana): queda, pues, como una insustituible obra de consulta.

LIBROS DE ESTILO EN C++

Tras superar la etapa introductoria y asimilar los recursos del lenguaje, lo siguiente es aprender a codificar en C++ con efectividad y limpieza. Los textos que se detallán a continuación pretenden cubrir la etapa intermedia de formación del programador de C++, recalando más que en la sintaxis en los esquemas conceptuales que subyacen bajo ésta.

Effective C++: 50 Specific Ways to Improve Your Programs and Designs, por Scott Meyers, 1992, Addison-Wesley, 0-201-56364-9.

Este libro es, en esencia, una colección de 50 consejos sobre programación en C++, debidamente comentados y justificados. Pero no se trata de un recetario al uso, sino más bien de un compendio de lineas maestras, interrelacionadas entre sí (como el lector pronto descubre) y tendentes a procurar al lector este tipo de conocimiento que subyace tras la seca sintaxis del lenguaje. Así, aunque el lenguaje permite usar con libertad la derivación

pública, la aproximación de éste al espíritu de OOD exige que tal derivación se use solamente cuando se da una relación de subtipo (o sea, de "ES-UN") entre las clases a relacionar. El lenguaje permite, también, por ejemplo, la redefinición en clases derivadas de funciones miembros no-virtuales de las clases base: esta práctica vulnera, sin embargo, como bien indica Meyers, la cualidad de coherencia de los programas: el comportamiento de un objeto variará dependiendo del puntero desde el que sea accedido. Como vemos, los casos estudiados son de enorme utilidad práctica. Junto a éstos se exponen técnicas que suelen pasar desapercibidas a los principiantes en C++, como, por ejemplo, el chequeo de auto-asignación en la implementación del operador de asignación de una clase. Así, nos encontramos en el índice con secciones como las siguientes: "Nunca redefinas el valor heredado de un parámetro por defecto", "Cualifica los destructores como virtuales en las clases base", "Chequea el valor de retorno de new", "Evita los datos miembros en el interfaz público", para terminar con "Lee el ARM". El libro, totalmente recomendable, proporciona al lector la inteligencia de cómo usar los recursos sintácticos de C++ asimilados de los textos introductorios clásicos. Además, su lectura es tan amena como la de una novela policial.

C++ Programming Style, por Tom Cargill, 1992, Prentice Hall, 0-201-56365-7

Junto con el libro de Meyers, configura una de las mejores inversiones en papel impreso que puedan realizar los ya iniciados en el lenguaje C++. El enfoque adoptado por Cargill es, no obstante, distinto al de aquél: en vez que acopiar reglas parciales y supuestamente independientes (como una shopping list), el libro reúne porciones de código extraídas de otros textos de C++ y de productos comerciales, para luego examinarlos con el ojo crítico del estilo. Es sorprendente la cantidad de elementos perturbadores, y aun de graves errores, que Cargill hace aparecer ante nuestros ojos, demasiado acostumbrados a las revisiones rutinarias y a los esquemas del mínimo esfuerzo. Se tratan, así, temas como "Herencia innecesaria", "Consistencia de clases", etc. analizando codificaciones de clases como "String", "Stack", "Máquinas de Estados Finitos", etc. Cargill pretende imbuir al lector de determinadas reglas de buen estilo en C++, tales como las siguientes: "Identifica el delete para cada new", "No uses un constructor para inicializar datos miembros estáticos", "Considera los argumentos por defecto como una alternativa a la sobrecarga de funciones", etc. Como se puede fácilmente apreciar, el texto no tiene desperdicio. El tipo de conocimiento que procura es, por otro lado, de una madurez matizadamente más clara que en el anterior texto, pues el lector adquiere una visión global de la cohesividad del estilo a imprimir al código. El mismo Scott Meyers reconoce su proximidad a este texto, indispensable a todo programador de C++.

C++ Programming Guidelines, por Thomas Plum & Dan Saks, 1991, Plum Hall, 0-911537-10-4, 274 pág.

Los que experimenten cierta aversión por las colecciones de reglas y los manuales de estilo, encontrarán en este libro el arquetipo de sus pesadillas. Al menos en el formato. Se trata, en efecto, de secciones codificadas que observan el siguiente esquema: TÓPICO (por ejemplo, 6.06 virtual_fct - funciones virtuales), REGLA (descripción de las líneas maestras de estilo correspondientes al tema del título), EJEMPLO (demostración práctica de la problemática contextual en que se desarrolla la regla), JUSTIFICACIÓN (argumentación de apoyo, tanto mediante razonamiento directo como a través de informes y textos externos, de la regla expuesta), ALTERNATIVAS (opciones de la regla consideradas menores, o aun impracticables), NOTAS LOCALES (media página al menos, y normalmente página y media, en blanco para el supuesto apunte de notas por el lector). Bien, decíamos que una tal rígida esquematización pudiera resultar demasiado restrictiva. El presente texto es, sin embargo, enormemente provechoso. Muy en la línea de un anterior libro de los mismos autores ("C Programming Guidelines"), sus distintas secciones tienden a reforzar una idea que los principiantes en C++ rápidamente relegan al olvido: C++ no es C. C++ no es, tampoco, Smalltalk o Eiffel. Se trata de sistematizar las soluciones a los cuellos de botella en la gestión de proyectos en C++, fundamentalmente cuando intervienen equipos de más de dos personas: a este fin se explicitan convenciones para nominar identificadores, facilidades para comentar código, gestión de sobrecargas, etc. El libro es, en definitiva, un compañero a tener muy en cuenta ante el diseño efectivo de clases realmente cohesivas y portables.

LIBROS DE PROGRAMACIÓN MEDIA Y AVANZADA EN C++

Este es un apartado singularmente despoblado en el actual panorama editorial sobre C++. No hay que dejarse engañar, por supuesto, por títulos como "Inteligencia Artificial con Turbo C++" o "Fabrique su propia red neuronal con Zortech C++ 3.0". Los tópicos avanzados del lenguaje constituyen, en la mayoría de los casos, un universo impensable por el lector de introducciones. En este caso, pues, la selección ha resultado evidentemente fácil. He preferido, con todo, reunir los textos bajo el epígrafe común "medios y avanzados" para evitar tener que calificar como avanzada únicamente a la obra de Coplien (mi sincera opinión). En fin: lo mejor será que el lector juzgue y establezca su propia ordenación.

A C++ Toolkit, por Jonathan S. Shapiro, 1990, Prentice Hall

Uno de las primeras necesidades de un desarrollador de C++ es la lectura y el estudio de código "práctico": tras una etapa de aprendizaje generalmente no todo lo corta que uno hubiera esperado, el principiante llega a estar

ciertamente hastiado de estudiar código general, pedagógicamente aceptable pero de nula o ligerísima aplicación en el mundo real. Este libro trata, precisamente, de los componentes reutilizables que con la frase "no reinventen la rueda" se suelen obviar en los textos introductorios de C++. La obra, resultando agradablemente breve, ofrece, tras una corta aproximación a los conceptos muy básicos de Orientación-a-Objetos y OOD, código de herramientas tan utilizadas como listas enlazadas, árboles binarios, arrays dinámicos, etc. El texto se acerca, desde la óptica del OOD, a la codificación de cada uno de estos componenentes, para ofrecer los listados completos en la última parte del libro. Se proporcionan, también, algunas líneas para mejorar distintos aspectos de los programas C++, aunque, por estar basado en AT&T C++ 2.0, no incorpora, como sería sobremanera deseable, plantillas (templates), supliéndolas mediante macros genéricas del preprocesador. El autor expresamente autoriza el uso comercial de los elementos software que aparecen en el libro, respetando el aviso de copyright, y de su calidad comercial dan fe distintas aplicaciones bien conocidas en las que aparece tal aviso. Con todo, y en contra de lo que el lector pudiera esperar, el texto no acompaña diskette, por lo que al posible usuario le esperan largas veladas de tedioso teclear. Naturalmente éste no es un libro introductorio al lenguaje, pues se supone que el lector ya conoce C++, sino, quizás, uno de los mejores candidatos para despegar de la etapa inicial de aproximación a C++.

Advanced C++ Programming Styles and Idioms, por James O. Coplien, 1992, Addison-Wesley, 0-201-54855-0.

Este es uno de esos escasos libros que redimen, con sobriedad textual y derroche de ingeligencia, el aluvión de tonterías y libros triviales sobre C++ y OOP que nos están inundando de forma inmisericorde. Al Stevens ha llegado a decir que este texto representa para C++ lo mismo que el de Kernighan & Ritchie representó para C: bien, quizá este comentario se limite a exteriorizar una pasión (fácil de comprender, por otra parte), porque sería más lógico aplicar tal comparación al texto de Stroustrup, pero ciertamente indica un cierto estado de ánimo que permanece al terminar el texto. Siendo este libro uno de mis preferidos, mi única recomendación es: ¡cómprenlo!. Pero, ¿de que trata esta obra? ¿De técnicas más o menos elaboradas de codificación en C++? Bien, no exactamente. Tras una breve, rigurosa y modélica introducción a los más interesantes tópicos del lenguaje, el autor nos anuncia de la existencia, dentro del lenguaje C++, de "idiomas" autónomos: esto es, de abstracciones por encima del nivel sintático elemental del lenguaje y que configuran, a partir de distintos supuestos, unas formas de codificar y un sustrato conceptual esencialmente distinto entre cada una de ellas. Recordemos, por ejemplo, que Stroustrup nos indica que los constructores en C++ no pueden ser virtuales, pero, a la vez, expresa que con facilidad se puede suplir tal característica. Coplien habla, sin embargo, de un "idioma de constructores virtuales". Tomemos como ejemplo este idioma para que el lector pueda entender mejor qué se esconde tras tal denominación. Un mensaje virtual permite posponer al tiempo de ejecución la ligadura entre prototipo e implementación de la función asociada; o sea, el mensaje se enviará a un objeto, desconocido en tiempo de compilación, y éste responderá de la forma más apropiada. Un constructor virtual supone lo siguiente: a un objeto, cuyo tipo exacto desconocemos, se le envía el mensaje "constrúyete", y éste eligirá el método de construcción apropiado pues, ¿quién habría de saber más de construirse que el objeto en sí?. Mediante la estructuración de una codificación que permita simular este comportamiento nos podríamos encontrar, por ejemplo, con que ninguno de los tipos de los objetos de nuestra aplicación sería chequeado en tiempo de compilación, pues el tipo exacto de un determinado objeto no nos haría falta ni siquiera para construirlo. Piense el lector que, de un plumazo, nos hemos "cargado" una de las cacareadas características de C++: el fuerte chequeo de tipos. Piense ahora el lector cómo sería codificar en C++ sin el chequeo de tipos: ¡muchas de nuestras ideas predeterminadas habrían de cambiar! ¡muchísimo de nuestro código ya no tendría sentido! Es como si, de repente, estuviéramos trabajando con un dialecto de C++: con un "idioma". Coplien describe algunos idiomas adicionales en su libro: "idioma de ejemplares", "idioma de estructuración en bloques", "idioma de lenguajes simbólicos", "idioma de herencia dinámica múltiple", etc. Ningún programador de C++ que aspire a algo más que a la codificación de una lista enlazada debería dejar de leer este texto. Hay que recabar, no obstante, que al lector se le supone un conocimiento adecuado de AT&T C++ 3.0, versión en que la obra está basada. Una última nota: los apéndices son realmente interesantes, y convienen en procurar al lector un agradable sensación de efectividad.

Data Abstraction and Object-Oriented Programming in C++, por Keith E. Gorlen, Sanford M. Orlow & Perry S. Plexico, 1990, John Wiley & Sons, 0-471-92346-X, 403 pág.

Este texto está basado en el estudio realizado por los autores para proveer de optimizaciones software a los sistemas UNIX de los Institutos Nacionales de Salud (NIH) en USA. Se trata, en esencia, de una librería de clases envolvente (conocida como librería NIH) modelada a imagen y semejanza del entorno Smalltalk-80 y prescindiendo de las capacidades gráficas de éste. El libro exige un solvente conocimiento previo de C++ y se apoya en AT&T C++ 2.0. Usando la terminología de Coplien, podríamos decir que aquí se genera un "idioma tipo Smalltalk", y lo cierto es que lo que en el texto se expone ha tenido una extraordinaria y reconocida influencia en el desarrollo de un gran conjunto de entornos de aplicación y librerías de clases genéricas. El provecho que los programadores de C++ pueden extraer de la obra es evidente, pues ésta está montada como un tutorial, explicando detalles y decisiones de diseño de las clases (algunas de ellas inapreciables para la adecuada construcción de componentes reutilizables, una de las expectativas más cacareadas y, a la vez, más difíciles de C++) y, tras esto, directamente utiliza el entorno creado como herramienta de desarrollo, explicando la posible extensión del sistema. La librería NIH incluye clases como Iterador, Bolsa, Diccionario, Tiempo, Vector, Fecha, Colección, Pila, Objeto (la raíz de la librería "cósmica", pues tal es el nombre por el que se conocen las librerías con una única clase base inicial). Como el lector puede fácilmente intuir, este tipo de librerías propende al usuario a practicar la derivación múltiple, y dadas las dificultades no siempre evidentes que esto entraña, el libro dedica un largo capítulo a este respecto. Se detalla, incluso, un ejemplo de aplicación de base de datos usando la librería. ¿Mi sugerencia? Bien, el fuerte entroncamiento de los conceptos de OOD con el acertado uso de C++ procuran un excelente texto de uso referencial, de valiosísima lectura para cualquier programador de C++ y aun de los estudiosos de OOP; las ventajas y defectos del enfoque tipo Smalltalk adoptado, por otro lado, disgustarán o deleitarán al lector, dependiendo en buena medida de su estilo y costumbres, pero esto aguzará, alternativamente, el ojo crítico o el gozo del lector. ¡Léanlo!

C++ Strategies and Tactics, por Robert B. Murray, 1993, Addi-son-Wesley, 0-201-56382-7, 273 pág.

Como el mismo título indica, Bob Murray, editor durante muchos años de C++ Report, ha querido mostrar, como en ajedrez, las estrategias y compo-nendas que trascienden los meros movimientos de las piezas del juego. No es exactamente, en general, un libro "avanzado" sobre el lenguaje, sino que más bien ocupa ese estadio intermedio entre los libros de estilo y la practicidad del código realmente efectivo. En este sentido los dos capítulos dedicados a las plantillas ("templates") justificarían, por sí solos, la lectura

del libro, pues dada la novedad comercial de esta adición al lenguaje, pocos textos se han ocupado hasta la fecha de revisar, de forma seria, algunos aspectos no triviales de la misma. Junto con las plantillas, el buen diseño de jerarquías de herencia en C++ y la reusabilidad del código ocupan el grueso de la obra, que, además, procura un capítulo sobre el manejo de excepciones. Nos encontramos, pues, ante una obra moderna y de potente claridad, pensada para el lector no-experto y que contiene ejercicios, resúmenes y una buena cantidad de ideas y sugerencias de buen diseño en C++. En definitiva, una adición indispensable para la biblioteca de C++.

LIBROS SOBRE SOFTWARE ORIENTADO-A-OBJETOS

El lector podría preguntarse aquí: ¿realmente necesito de textos sobre ideas generales de la orientación-a-objetos? ¿Qué ayuda me pueden prestar tales ideas en los procesos diarios de codificación en C++? Bueno, recordemos que C++ es un lenguaje con facilidades para la Programación Orientada-a-Objetos, y que tal es, en definitiva, la fase de implementación, tras las fases de diseño y análisis orientados-a-objetos, de los objetos y sus relaciones modelados en base a los conceptos de orientación-a-objetos a través de los que se matiza y visiona nuestro problema en el mundo real. Las ideas generales de este nuevo paradigma nos pueden ayudar, normalmente de forma inestimable, a encauzar nuestras codificaciones hacia modelos conceptuales más adecuados a la nueva orientación, consiguiendo que nuestro software sea más robusto y fiable.

Object-Oriented Software Construction, por Bertrand Meyer, 1988, Prentice Hall, 0-13-629031-0,534 pág.

El presente trabajo es, sin duda, la más rigurosa, acertada e inteligente exposición de los principios en los que se sustentan los sistemas orientados-a-objetos. A la vez, y esto puede desconcertar al lector inadvertido, es una primera descripción del lenguaje de programación Eiffel, del que el Dr. Meyer es directo arquitecto y creador. Pero empecemos con método. La primera parte, "Ideas y Principios", es absolutamente impagable: lo que en ella se expone, sustanciado en cinco criterios y seis principios, es frecuentemente usado por mí mismo en buena parte de los cursos de OOA & OOD que imparto. La exposición es sorprendentemente concisa, de forma que la revisión de conceptos supuestamente conocidos por el lector se torna en extremo interesante. Tras este análisis general de los sistemas orientados a objetos, el Dr. Meyer se cuestiona por un lenguaje con facilidades para su implementación, y como quiera que, según sus propias palabras, no encuentra ninguno, decide crear el suyo propio, sujeto con exactitud al paradigma de objetos. Aparece así Eiffel, lenguaje orientado a objetos puro donde los haya, de sintaxis tipo Pascal e interesantísimas características. Pero, ¿interesa esto al programador de C++? Así lo creo. El Dr. Meyer no se limita a describir el lenguaje, sino que, como artífice del mismo, explica las disyuntivas en las decisiones de diseño y justifica las medidas adoptadas en cada caso, trayendo a colación interesantes problemas presentes en muchos de los diseños orientados-a-objetos. Como ejemplo sirva el tratamiento que en el lenguaje se dan a lo que se denominan "precondiciones" y "postcondiciones": su claridad conceptual ha redundado en que, en aras de la modularidad, limpieza y coherencia del código, tal enfoque haya sido posteriormente adoptado por distintas librerías de C++, como, verbigracia, "Tools.h++" de Rogue Wave. Una tercera parte del texto se ocupa de revisiones genéricas de lenguajes clásicos de programación, así como de sus extensiones a objetos, y aun de C++, Smalltalk, Ada, etc. Se revisan también cuestiones de herencia y, de forma leve, cuestiones como la persistencia de objetos, que en el momento de publicación de esta edición no estaban todavía en el ojo del huracán, como ocurre ahora. En los apéndices se retoma, por fin, el lenguaje Eiffel a modo de fragmentos referenciales. Se trata, en resumen, de un libro indispensable para cualquier con pretensiones mínimamente serias en el ámbito de la OOP: reserven, pues, un hueco en su estante para él. O mejor aún: para su segunda edición, ya disponible en estos momentos.

A Book of Object-Oriented Knowledge, por Brian Henderson-Sellers, 1991, Prentice Hall, 0-13-059445-8, 297 pág.

El presente libro podría ser considerado como una eficaz introducción al panorama general de orientación-a-objetos y, aunque, según informa el propio autor, el texto pretende ser una guia en el proceso de formación y aclimatación de la mentalidad de los lectores al paradigma de objetos (para lo que incluye modelos de transparencias "ad hoc" que pueden ser copiadas y utilizadas como ilustración de cursos sobre la materia), la obra se constituye, en el fondo, en una esclarecedora revisión, desde una adecuada distancia (y recuérdese aquí la frase de Ortega sobre Cleopatra), de los distintos criterios que pueblan, a veces en descorazonador desorden, el universo de los objetos. Se revisan, así, metodologías de análisis, diseño e implementación, intentando procurar al lector una suerte de plataforma conceptual básica desde la que pueda acceder con mayor comodidad a técnicas concretas.

Object-Oriented Methods, por Ian M. Graham, 1991, Addison-Wesley, 0-201-65621-8, 410 pág.

Es éste un libro caracterizado por la perspectiva globalizadora bajo la que se contemplan las ideas y conceptos de orientación-a-objetos. La impronta pragmática británica se deja notar, y su lectura es realmente amena. El texto comienza con una bien entramada introducción crono-sectorial a los tópicos de la OOT: conceptos básicos, lenguajes de programación orientados-a-objetos, WIMPs, bases de datos relacionales, bases de datos orientadas-a-objetos, etc.; hasta llegar a la parte más significativa: análisis y diseño orientados-a-objetos. Aquí Graham, tras una revisión crítica de algunos métodos (HOOD, Coad/Yourdon, etc.), expone la metodología desarrollada en BIS Applied Systems: SOMA, una variación de Coad/Yourdon a la que se han añadido, simplificando, "triggers". La orientación-a-objetos es filtrada a través de la experiencia del autor en los campos de inteligencia artificial e ingeniería del conocimiento y, así, resulta curiosa y enormemente instructiva, por ejemplo, la facilidad con que Graham aborda la fase de identificación de objetos y de sus relaciones. La prototipación aparece descrita, seguidamente, de una forma esclarecedora, para terminar con un vistazo al futuro posible y un muy interesante apéndice sobre "objetos borrosos". En fin, se trata de un texto muy aconsejable para aquéllos que busquen una visión integradora de las nuevas técnicas en el continuum de la evolución informática. No está de más, al fin, probar un poco de solidez europea frente a las montañas (magníficas, por otro lado) norteamericanas.

Object-Oriented Programming, por Peter Coad y Jill Nicola, 1993, Prentice Hall-Yourdon Press, 0-13-032616-X, 582 págs. y disquete incluido.

Bueno, en la misma tónica que otros libros firmados en colaboración por Peter Coad, es éste un texto divertido, ameno y claramente pedagógico. Tras el índice, de increíble longitud, el lector se encuentra con una obra estructurada en cuatro partes, sustanciadas cada una de ellas en un ejemplo completo y en orden creciente de dificultad: "Contador", "Una maquina expendedora", "ventas, ventas, ventas" y "Sigue el flujo", seguido por apéndices en que se detallan las características esenciales de los lenguajes en que se desarrollan tales ejemplos: C++ y Smalltalk. Para el análisis y diseño de éstos se sigue, cómo no, el método y la notación de Coad/Yourdon, y cada paso es suficientemente explicado. La comparación de los ejemplos en C++ y Smalltalk sirve, de paso, como incruenta introducción a ambos lenguajes. Lo cierto es que el libro imbuye fácilmente al lector en el *object-thinking*, acostumbrándolo a pensar "en objetos", y sólo por esto valdría la pena leerlo. Aparecen, además, situaciones y decisiones de gran valor pedagógico. La campaña de comercialización del libro incluye "El Juego de los Objetos", con pelotita, silbatos y fichas de cartón, junto con un vídeo en el que se puede apreciar la vitalidad yanqui de Peter Coad.

LIBROS DE ANÁLISIS Y DISEÑO ORIENTADO-A-OBJETOS (OOA & OOD)

A pesar de la apariencia de complejidad que normalmente se suele aplicar a estas fases del proceso de desarrollo software, lo cierto es que, como resulta de mi propia experiencia impartiendo cursos diferenciados de OOP, OOA y OOD, aunque normalmente se empieza por el lenguaje, cuando se revisan los métodos de análisis y diseño Orientados-a-Objetos, se hace sentir entre los alumnos una fuerte sensación de que, comprendidas las bases de estas disciplinas, se pueden aprovechar mejor los recursos de C++. No estoy propugnando que el lector escoja necesariamente una de las metologías presentes en el mercado, sino que examine el sustrato en el que se apoyan y se apropie de alguna de las técnicas que en ellas aparecen.

Designing Object-Oriented Software, por Rebecca Wirfs-Brock, Brian Wilkerson & Lauren Wiener, 1990, Addison-Wesley, 0-13-629825-7, 368 pág.

Este es un texto absolutamente recomendable, sin contrapartida alguna, a aquéllos que buscan iniciarse en las procelosas aguas del Diseño Orientado-a-Objetos. Esto no significa, empero, que se trate de un libro elemental: de hecho, la solidez conceptual en la que se apoya convierte al texto en una muy fructífera fuente y base referencial para profesionales y equipos con experiencia en Tecnología de Objetos. Esta obra intenta, en síntesis, proporcionar un método de OOD independiente de lenguajes y aun de notaciones especiales. Dado que el autor del presente libro es especialmente sensible a lo que se conoce como diagramanía (esa costosa, compleja y contraproducente acumulación de dibujos y conexiones,

escasamente diferenciados entre sí, y que suelen conducir a analista, diseñador, cliente y, en general, a cualquiera que inocentemente los examine, a un estado de perplejidad cercano al catatónico), la exposición que aquí se realiza, capitaneada por Wirfs-Brock, es como una burbuja de oxígeno en la luna. El grueso del libro trata de lo que otros, en esta misma materia, dan mayormente por supuesto o examinan muy brevemente: la identificación y asignación de objetos y de sus interrelaciones. Para el tratamiento de la información relacionada con el proceso de OOD se usan las denominadas fichas CRC (Clase-Responsabilidad-Colaboración): en cada una de ellas se significa el nombre de la clase, si es abstracta o no, las superclases y las subclases de la misma, una sucinta descripción de su cometido, las responsabilidades que asume (ese suma de un cierto tipo de conocimiento que la clase posee y las acciones que puede efectuar), y las colaboraciones necesarias para cumplimentar cada una de sus responsabilidades (esto es, la relación de clases necesarias para llevar a cabo las tareas o responsabilidades asignadas y para las que la clase no es autosuficiente). Se trata en síntesis del siguiente proceso: en una primera etapa exploratoria se produce la identificación de clases, identificación de clases abstractas, identificación y asignación de responsabilidades e identificación de colaboraciones; en la siguiente etapa, denóminada de análisis, se modela la construcción de jerarquías (clases en derivación), se identifican los contratos (una serie cohesiva de responsabilidades normalmente requeridas por otras clases) y se asignan a las colaboraciones, se identifican los subsistemas (abstracciones que permiten un manejo más adecuado del sistema global a modelar), se refinan las colaboraciones y se protocolizan las responsabilidades (esto es, cada una de las responsabilidades se transforma en el prototipo de una función). Se obtiene, al final, algo así como una estructura de clases y sus relaciones vacía de implementación. Pero es que la implementación no es lo importante: la identificación de las responsabilidades (por servicios) públicos de las clases permitirá fragmentar la implementación de clases de una manera adecuada para el trabajo en equipo (una clase o serie de clases pueden serle asignadas a una persona, por ejemplo, y ésta únicamente sabrá del resto de las clases que pueden ser accedidas a través de un protocolo público ya bien definido, independientemente de su implementación concreta). El seguimiento del clásico ejemplo del cajero automático es particularmente revelador sobre el proceso de diseño expuesto, con abundantes comentarios y la explicación detallada de las decisiones tomadas. Este ejemplo, y otros, están expuestos en su totalidad en los apéndices del libro. Expuesto lo anterior, reitero mi recomendación de uso de este libro quizás como el primer libro de OOD que los principiantes debieran estudiar.

Object-Oriented Modeling and Design, por James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy & William Lorensen, 1991, Prentice Hall, 0-13-629841-9, 528 pág.

En este libro se expone una metodología de análisis y diseño orientados a objetos, que abarca el ciclo completo de desarrollo de software, creada por el equipo de investigación de General Electric liderado por James Rumbaugh. De todas las metodologías propietarias examinadas aquí, ésta es una de las más completas y, a la vez, la que menos reniega de las bien conocidas metodologías de análisis y diseño estructurados en las que abiertamente se basa. Los autores plantean el traslado del dominio del problema en el mundo real al campo software a través del modelado de los objetos que aparecen en aquél, junto con sus relaciones. Y aunque esto mismo podría decirse de cualquier otro técnica de OOA&OOD, el método aquí expuesto, denominado OMT (Object Modeling Technique: Técnica de Modelado de Objetos), se descompone en tres submétodos: el modelado de objetos, el modelado dinámico y el modelado funcional. Examinemos, por su interés, aun de forma sucinta, estas tres subtécnicas: en el modelo de objetos se describen, con una notación clara y precisa, los atributos y las relaciones estáticas entre los clases a que pertenecen los objetos del problema y sus relaciones ya modeladas; en el modelo dinámico se exponen, mediante diagramas de transición de estados, los aspectos del sistema relacionados con el control del esquema secuencial y temporal de las operaciones que afectan a los objetos modelados, usando, para mayor facilidad en la comprensión del diagrama, la abstracción de Harel que permite la anidación de subsistemas de estados; en el modelo funcional se exponen, por último, los aspectos del sistema relacionados con las transformaciones de la representación interna de los objetos modelados, ya expresadas estáticamente como operaciones en el modelo de objetos, a través de los muy conocidos diágramas de flujo de datos. Quizá el modelado funcional resulte el de peor acoplamiento en las bases conceptuales de un esquema de objetos, pero, aun así, su interrelación con las demás técnicas es modélica. Estos tres modelos se engarzan a través de lo que se pretende -o mejor, adivina- sea un esquema no secuencial de fases: análisis, diseño del sistema y diseño de objetos. La exposición final de ejemplos (un compilador de diagramas de objetos, animación computerizada y un sistema de diseño de distribución eléctrica) es un excelente complemento de esta técnica. Estamos, en definitiva, ante un excelente texto, aderezado con comentarios más o menos acertados sobre bases de datos relacionales y con extensiones a objetos, lenguajes no orientados-a-objetos, etc. Se aprecia, por último, una importante característica de cohesividad a lo largo de la exposición de la metodología que le proporciona una solidez conceptual a que otras son ajenas. La OMT ha ido ganando, desde su publicación, adeptos entre los usuarios de OOA&OOD, habiéndose posicionado, en estas fechas, como una de los métodos de uso más extendido. Se utilice o no esta técnica, lo cierto es que el libro es, en todo caso, una valiosísima contribución a cualquier biblioteca de OT. General Electric ha desarrollado una herramienta denominada OMTool para distintas plataformas que computeriza estas técnicas.

Object-Oriented Design with Applications, por Grady Booch, 1991, Benjamin/Cummings, 0-80353-0091-0, 580 pág.

Este libro, muy en la línea de los dos anteriores, es uno de los más considerados por la comunidad C++. Abarca únicamente la etapa de diseño, no recogiendo las últimas adiciones a lo que se denomina notación Booch. Constituye la perfecta continuación, con un carácter marcadamente propio, del texto de Wirfs-Brock. Como la mayoría de los textos en OOA y OOD, empieza por una revisión de los conceptos básicos de la orientación a objetos, tales como clases, objetos, etc., exponiendo distintos mecanismos para la identificación de estos dentro del dominio de un problema. Seguidamente se expone el método de OOD de Booch, junto, cómo no, a su correspondiente notación (donde aparecen las famosas nubes, que el lector posiblemente habrá visto en alguna ocasión). Por último se detallan suficientemente distintos ejemplos aplicativos de la metodología expuesta, implementados en CLOS, C++, Smalltalk, Ada y Object Pascal. Un apéndice sobre lenguajes de programación orientados y basados en objetos, además de 46 páginas repletas de bibliografía completan el volumen. Nos encontramos, pues, ante uno de esos raros textos densos en contenido y, a la vez, de brillante practicidad y fuertes capacidades referenciales. Piense el lector que uno de los "reproches" que se le imputan a Booch es la "excesiva riqueza" de su notación. ¿Mi consejo? Este debe ser el segundo o, a lo sumo, tercer libro de diseño que adquieran. Existe una herramienta denominada Rose, de la compañía Rational (la misma a que pertenece Grady Booch), que soporta esta metodología.

Object-Oriented Analysis, 2nd Edition, por Peter Coad & Edward Yourdon, 1991, Yourdon Press/Prentice Hall, 233 pág.

Nos encontramos ante la segunda edición de uno de los primeros textos aparecidos en el mercado sobre OOA. Esta premura editorial consiguió que las ideas propuestas por Coad y Yourdon se extendieran con gran rapidez, siendo así que este método ha permanecido durante mucho tiempo como uno de los más ampliamente difundidos. El libro es uno de los más breves de los comentados en este anexo y está escrito en un lenguaje coloquial de muy fácil lectura. Los conceptos básicos del paradigma de objetos se exponen con la ayuda de definiciones de diccionarios y enciclopedias. La graficación, una de las primeras y, por tanto, de las más rudimentarias en este campo, básicamente expone el resultado gráfico de la herramienta comercializada por los autores (OOATool). Las secciones de introducción y comparación de distintas metodologías de análisis anteriores al OOA (Yourdon-de Marco, Jackson, etc.) son agradablemente sucintas y claras. Lo único que se puede reprochar es la falta de un formalismo metodológico práctico que permita al lector hacer uso de lo aprendido. De hecho el lector se queda al final del texto con una cierta sensación de borrachera de objetos, generada en buena medida por el énfasis y la excitación con que los mismos autores tratan a la OT, pero sin direccionamiento práctico claro en el que sostener sus primeros pasos en este campo. El libro es, pues, perfectamente aconsejable como texto introductorio a OOA.

Object-Oriented Design, por Peter Coad & Edward Yourdon, 1991, Prentice Hall, 0-13-630070-7.

Estamos ante una clara continuación del anterior libro de los autores sobre OOA, que usa de una extensión apropiada a OOD de la herramienta de análisis OOATool de los autores. Tras una leve introducción (pues el texto, como el anterior, es singularmente corto), se repasa la metodología de Análisis orientado-a-objetos de Peter Coad: un modelo multicapa (sujeto, clase-objeto, structura, atributos y servicios) y multicomponente (dominio del problema, interacción humana, gestión de tareas y gestión de datos), con una notación específica de aplicación. Los autores intentan, con cierto éxito, integrar de forma incruenta las técnicas de OOD con las del proceso de OOA, para pasar después a exposiciones sobre sectores de parcial interés, como el de las herramientas CASE o los distintos lenguajes de programación. El texto, con las mismas salvedades de la obra anterior, es totalmente recomendable como introducción no reglada al campo del OOD.

Object-Oriented Analysis: Modeling the World in Data, por Sally Shlaer & Stephen J. Mellor, 1988, Yourdon Press/Prentice Hall, 144 pág.

Tenemos aquí a uno de los pocos textos serios focalizados en el área de análisis orientado-a-objetos. Con un corto número de páginas, la obra comienza con una simpática introducción a los típicos conceptos básicos, pasando a poco a clasificar los objetos en: tangibles, roles, interacciones, incidentes y especificaciones (he de reconocer que esta fragmentación conceptual yo siempre la he asumido como de identificación de clases). Seguidamente expone un modelo de control textual de especificaciones de clases como soporte de la técnica desarrollada por los autores y denominada modelado de información. No es éste un libro que proporcione un bagaje semejante al de Wirfs-Brock, ni la técnica en él descrita es particularmente fácil de aplicar, pero, con todo, el texto ofrece detalles muy interesantes para el estudioso, así como ejemplos altamente intuitivos y de ilustraciones autoexplicativas.

Object Lifecycles: Modeling the World in States, por Sally Shlaer & Stephen J. Mellor, 1991, Prentice Hall, 0-13-629940-7.

Esta obra estudia, tras un breve repaso comprehensivo de la anterior obra de los autores, el comportamiento dinámico de los sistemas de objetos. Éstos se asimilan a máquinas de estados (así como las clases a modelos de estados), de tal forma que el ciclo de vida de un objeto puede modelarse como un conjunto de estados, de eventos, de reglas de transición y de acciones. Seguidamente se muestra el desarrollo de las relaciones entre objetos afectadas por el tiempo, para pasar después a la exposición de los métodos de modelado de secuenciaciones de eventos y terminar con una extensión de los diágramas de flujos de datos denominada ADFD y referida a los datos asociados a acciones de los objetos. Se exponen diversas posibles aplicaciones de la técnica de la obra y se muestran algunas líneas de migración a esta metodología desde el enfoque estructurado. En definitiva nos encontramos ante el perfecto compañero de la anterior obra.

Object-Oriented Systems Analysis: A Model-Driven Approach, por David W. Embley, Barry D. Kurtz & Scott N. Woodfield, 1992, Prentice Hall, 0-13-629973-3, 302 pág.

He aquí, a mi entender, una de las aportaciones más significativas realizadas en los últimos tiempos al área del análisis orientado-a-objetos. Los autores proveen, a más de una definición formal de su método de análisis (OSA: Object-Oriented Systems Analysis) basado en lo que llaman ORM (Object-Relationship-Model), de la que carece el resto, una nueva y rica notación que se aplica con exactitud al nuevo enfoque de aproximación al OOA: relaciones, estados y modelos de interacción de objetos. Si bien los modelos de relación y de estados son fácilmente asimilables (salvadas ciertas

distancias) desde otras metodologías (diagramas entidad-relación, etc.), el modelo de interacción reemplaza al clásico diagrama de flujo de datos. El tratamiento dado, de cualquier forma, a las relaciones es particularmente rico y revelador: éstas aparecen ya como objetos reales, y la notación cualificadora de los conjuntos de relaciones resulta increíblemente clara, en perfecto desarrollo de aproximaciones más tradicionales como la de Rumbaugh (OMT). La obra contiene una gran cantidad de ejemplos y cuestiones (existe un libro adicional con la respuesta a los ejercicios propuestos) sencillamente perfectos. El capítulo introductorio es, por otro lado, suficientemente explicativo, y el ejemplo del trayecto en la ciudad es realmente bueno. Bien: su biblioteca quedaría incompleta sin esta obra. Así de simple.

Object Oriented Program Design with Examples in C++, por Mark Mullin, 1990, Addison-Wesley, 0-201-51722-1, 303 pág.

Este texto ha llegado a ser muy popular entre un cierto sector de la comunidad C++, principalmente entre los desarrolladores provenientes de C y de esquemas de diseño estructurado, debido quizás a la inmediata practicidad de los conceptos que, sin esquematización rigurosa, se van proponiendo: es como si se diseñara en C++. El autor asume un ejemplo (la creación de una base de datos corporativa para Bancroft Trading Company) y en sucesivos capítulos va refinándo su diseño. La mayor ventaja para algunos de este libro (su proximidad a la vida real) es, sin embargo, también su principal defecto. Hay que pensar que en un esquema serio de OOD, por ejemplo, el diseño de bases de objetos (por bases de datos orientadas a objetos, o simplemente de datos) **no existe**. Efectivamente: los modernos gestores de bases de objetos procuran, mediante la no diferenciación entre medios de almacenamiento primario y secundario, una suerte de memoria virtual infinita para nuestros objetos: no hay que preocuparse de extensiones del lenguajes, preprocesadores o procedimientos de archivo y recuperación de objetos, así como tampoco de convertir los punteros en identificadores, o de descomponer los objetos en tablas. En realidad, en la mayoría de las implementaciones en las que aparece un OODBMS, los objetos son persistentes por defecto, a menos que expresamente el desarrollador indique lo contrario. De acuerdo, perfecto, pero ¿a qué viene esta perorata? Bueno, si no hay diseño que hacer de bases de datos, el libro entero de Mullin queda un tanto desenfocado, pues la importancia habría de traspasarse a la identificación de clases y relaciones, algo que en el texto se relega a un segundo plano. Planteada esta observación, por lo demás el libro es recomendable como primer estadio de paso en la codificación C++ basada en la conceptualización de objetos, sin demasiadas pretenciones adicionales.

A Complete Object-Oriented Design Example, por Joseph E. Richardson, Ronald C. Schultz & Edward V. Berard, 1992, Berard Software Engineering Inc, 1-881974-01-4, 350 pág.

Se suele echar de menos, al introducirse en el terreno del OOD, la disponibilidad de un ejemplo completo, en todas su fases, que rellene esas carencias, obviadas por una cuestión esencial de espacio en otros textos, y que colocan al novicio en disyuntivas normalmente difíciles de superar. El presente texto intenta cubrir tan urgente necesidad. El lector fácilmente podrá apreciar que en el libro se explicitan, con toda clase de detalles, las fases consideradas más farragosas y de trabajo más tedioso, normalmente sustanciadas en listas selectivas: listas y más listas. Se usan, después, diagramas de transición de estados, redes de Petri y diagramas de Booch para resolver el problema de diseño de una utilidad de concordancias, típica de un procesador de textos. Finalmente se detalla la implementación completa de la solución encontrada, tanto en Smalltalk como en C++. En el apéndice se detallan distintas especificaciones para terminar con una exposición parcial de las diapositivas empleadas por la firma de ingeniería de software, editora del libro, en sus cursos y trainings de OOD.

A-3

OOA & OOD: UNA APROXIMACIÓN CRÍTICA

El propósito de este anexo es mostrar, en primer lugar, el sustrato conceptual en que se basan los sistemas software orientados-a-objetos y que, a la vez, anima las variadas metodologías que cubren las distintas fases del desarrollo de software. Seguidamente se practicará una sucinta aproximación crítica a parte de las técnicas que pueblan los distintos métodos en las áreas de análisis y diseño orientado-a-objetos, con un especial énfasis en el campo del diseño, para terminar con una exploración de los distintos textos y herramientas comerciales que permiten la aplicación de tales metodologías. El tono del texto pretende ser decididamente globalizador, evitando, en lo posible, la introducción de matizaciones conceptuales o terminológicas que aumenten la aparente confusión formal en las áreas citadas.

SOFTWARE ORIENTADO-A-OBJETOS

El paradigma de la orientación-a-objetos filtra, matiza y modela convenientemente cada una de las fases del proceso de mapeo de un determinado problema real a un sistema software. Podríamos decir, pues, que el sistema software resultante, producto de la sistemática aplicación de criterios y fundamentos bien determinados y diferenciados, ineludiblemente habrá de constituirse en un Sistema Software Orientado-a-Objetos (OOSS: Object-Oriented Software System). Ahora bien, si se piensa que las fases del proceso de desarrollo son las bases en que se apoya el OOSS, verdadero fin de aquél, debemos preguntarnos: Realmente, ¿qué se quiere construir? ¿Qué es un OOSS? O mejor, ¿qué caracteriza de forma general a un OOSS?

Pero, ¿estamos planteando la pregunta apropiada? Sin duda: una respuesta acertada restringirá y clarificará los caminos de acceso al fin propuesto. De hecho, otro tipo de pregunta que cuestionara únicamente las diferencias con relación a los SSS's (Structured Software Systems) nos habría de conducir a

la equivocada vía de considerar la Orientación-a-Objetos como una mera adenda al paradigma estructurado. De acuerdo con esto, lo primero sería intentar establecer las principios básicos caracterizadores de un OOSS, y, para esto, debemos examinar lo que a este respecto opinan distintos autores. En [Hende90] aparece un cuadro de especial interés, por cuanto que evidencia la disparidad de criterio, aún no consensuada en la actualidad, sobre lo que en esencia define a un OOSS: ocultación de información, encapsulación, objetos, clasificación, clases, abstracción, herencia, polimorfismo, ligadura dinámica, persistencia y composición. Cabe notar, no obstante, la fuerte concordancia en la "herencia" como una característica básica, aunque es mi opinión -coincidente con Winblad et al, Bloy et al y Henderson-Seller- que la herencia es más un mecanismo básico que un concepto básico. Pensemos, por ejemplo, que en C++ la derivación (por herencia) es el mecanismo que conviene al lenguaje características polimórficas. Hay que ponderar, también, que existen otros mecanismos, como el de delegación, sustitutivos de la herencia. De esta manera, Henderson-Sellers establece lo que denomina "Triángulo de la Orientación-a--Objetos", en el que se establecen los pilares básicos de los OOSS: abstracción, encapsulación y polimorfismo. Naturalmente, esta figuración es resultado de un siempre arriesgado truncamiento de opciones, por lo que es tan discutible como las aproximaciones en que se basa.

Realmente es difícil encontrar, en la cada vez más profusa -y confusa- relación bibliográfica de orientación-a-objetos, una rigurosa caracterización extensiva de los OOSS. Las características notadas en el párrafo anterior parecen, de hecho, corolarios de conceptualizaciones de más bajo nivel en la comprensión de tales sistemas software. Una de las pocas revisiones formales de los OOSS's la encontramos, sin embargo, en [Meyer88], que afirma, tras establecer la **modularidad** como objetivo esencial en los sistemas software, que ésta habrá de estar animada y regida por cinco criterios y seis principios esenciales. Por su especial interés, procederemos a revisarlos con más detalle, pues mediante el examen de su cumplimiento en un determinado sistema software podrá determinarse, con la vaguedad formal propia de este paradigma, el grado de orientación-a-objetos de éste. Meyer no pretender describir una metodología, pero, como se pondera en [Rumba91], ofrece muy buenas técnicas de buen diseño.

CINCO CRITERIOS SOBRE MODULARIDAD EN OOSS

Criterio de Descomponibilidad Modular: Un OOSS cumplirá este criterio si facilita la fragmentación de un problema en varios subproblemas o, en otro nivel conceptual, favorece que los módulos de que se compone sean particionables en submódulos, y estos a su vez en otros, hasta llegar a un nivel que permita aprehender y abordar su tratamiento individual. Naturalmente la típica estructuración top-down cumple la condición descrita, aunque, en realidad, hay que considerarla como una muy particular (e inmediatamente eficiente) interpretación del criterio, como un modelo en base a diagramas de Venn rápidamente evidencia: la libertad de descomposición se troca aquí en forzosa estructuración arbórea.

Criterio de Componibilidad Modular: Se trata aquí del grado de facilidad con que pueden combinarse de forma flexible (aunque no siempre con total independencia) los módulos de que se compone un OOSS. Este criterio está directamente relacionado con la posibilidad de reutilización de componentes para la creación de nuevos sistemas software en diferentes entornos y escenarios. De hecho, en justa correspondencia, la reutilización ha sido definida como "la capacidad efectiva de incorporar objectos creados para un sistema software dentro de un sistema software diferente" ([Wasse91]). Hay que tener en cuenta, empero, que el criterio de descomponibilidad no conduce necesariamente a éste, pues la mera descomposición suele abocar -como ocurre en el enfoque top-down- a módulos no reutilizables (cuestionémonos, si no, cómo podríamos "recomponer" a una persona viva anteriormente descuartizada: ¿quizá con un módulo Shelley?). La idea que, en definitiva, subyace en las ideas expuestas es la del uso de distintos repositorios de módulos (quizás modelados como OODBMS's) que permitan la construcción independiente de sistemas software.

Criterio de Comprensibilidad Modular: De poco sirve una estructura o configuración modular si ésta no puede ser parcialmente asignada o comprendida por un observador externo. Si atendemos, por otro lado, a la consideración de G.I.Miller de que el límite humano medio para el procesamiento de información, en el marco de la memoria inmediata, se situa en tres distintos conjuntos conteniendo un máximo de tres ítems cada uno, la necesidad de poder extraer de un OOSS un pequeño subsistema comprensible para el lector se torna aún más perentoria. El presente criterio ilustra, pues, la necesidad de que los módulos en un OOSS sean autoexplicativos o, a lo sumo, extiendan su comprensibilidad a alguno de los módulos adyacentes.

Criterio de Continuidad Modular: De forma intuitivamente parecida a como ocurre con la continuidad de funciones y=f(x), donde informalmente puede decirse que un pequeño cambio en x produce un necesariamente pequeño cambio en y, subyace bajo este criterio una idea que se ha totemizado, justamente, en los manuales de estilo de OOP: "pequeños cambios han de originar pequeñas repercusiones". Un ejemplo práctico de este criterio lo

constituyen las facilidades polimórficas de la ligadura dinámica, que permiten la incorporación de un nuevo modulo a un OOSS sin apenas cambios que afecten a los demás: así en C++, mediante el mecanismo de funciones virtuales, pueden añadirse al sistema objetos de clases nuevas, sometidas a una relación jerárquica de derivación pre-establecida, sin modificar en absoluto el esquema de mensajes y relaciones de tipo entre clases.

Criterio de Protección Modular: Se pretende que la propagación en tiempo de ejecución de un error en un módulo resulte local a éste o, como máximo, se extienda a algunos de los módulos adyacentes.

SEIS PRINCIPIOS ESENCIALES EN OOSS's

Unidades Modulares Lingüísticas: De acuerdo con los criterios anteriormente establecidos, es imposible pensar en módulos a la vez descomponibles, componibles, comprensibles, continuos y protegidos contra la propagación de errores si éstos no representan entidades conceptuales con límites sintácticos bien definidos. La encapsulación de información en módulos sin tales fronteras semánticas deviene desafortunadamente frágil, apropiada quizás para un escenario específico, pero de difícil o imposible extrapolación a otros. La aplicación de este principio se ha convertido, difundida en la base de distintos métodos de OOA & OOD, en uno de los criterios evaluadores de la idoneidad de los módulos identificados en las distintas fases del ciclo de vida de un OOSS: "Si a un candidato a módulo, clase o subsistema no se le puede aplicar una unidad linguística del lenguaje en que estemos operando, es momento de reconsiderar las decisiones que condujeron a su elección". Como quiera que la obra de Meyer está fuertemente orientada a la imposición de un lenguaje como expresión visual del diseño, llega a concluir, a diferencia del posicionamiento general de autores de otras metodologías de orientación gráfica, que la implementación de estos diseños no podrá ser abordada por lenguajes que carezcan de tales facilidades modulares sintácticas.

Pocos Interfaces: Un módulo debe comunicarse con tan pocos otros módulos como sea posible. En términos de Wirfs-Brock, la "inteligencia" de un sistema debe distribuirse entre los módulos de forma local a éstos. Lo que se pretende, al fin, es liberar al propio sistema bien de módulos profusamente interconectados entre sí (vulnerando la totalidad de los criterios expuestos) bien de módulos que encierren la total "inteligencia" del mismo y que, en tal razón, deban comunicarse, en claro esquema centralista, con la mayoría de los demás.

Interfaces Pequeños: Un módulo debe intercambiar con los otros módulos tan poca información como sea posible. Naturalmente la restricción de opciones, habitualmente concretada en la limitación de los mensajes a que un módulo puede responder, ayuda sobremanera al cumplimiento de este

principio. De hecho, su más extendida violación se sustancia en las interminables listas de argumentos entre módulos, normalmente debidas al malentendido uso de librerías, expresamente contrarias a los criterios de continuidad y protección modular.

Interfaces Explícitos: Se enfatiza aquí la necesidad de que la comunicación entre módulos sea inmediatamente evidente al observador externo. ¿Por qué? Bueno, es ciertamente difícil reutilizar, trasladando de un escenario a otro, módulos que mantienen relaciones sutiles o subrepticias con otros módulos o subsistemas, pues éstas no habrán sido tenidas en cuenta y bien serán cercenadas de forma peligrosa, bien originarán indeseables efectos laterales difíciles de depurar. De alguna manera el interfaz del módulo es el equivalente a la ficha policial de éste: no se desean sorpresas. Meyer sintetiza con acertado grafismo la aplicación de éste y los dos anteriores principios: es como si se instaurara la dictadura en el universo de los módulos, pues a estos no se les deja reunirse en grupos numerosos, se les obliga a cruzar tan sólo algunas palabras entre ellos y, además, se les fuerza a hablar gritando. Quizá una aproximación similarmente didáctica consista en considerar a los módulos como psicópatas en libertad vigilada.

Ocultación de la Información: como directa consecuencia de la encapsulación, cada módulo esconde sus datos de otros módulos de manera que éstos sólo podrán ser accedidos a través de los métodos del propio módulo. Algunos autores, como Meilir Page-Jones, prefieren hablar más bien de "ocultación de la implementación", mientras que otros, como Booch, directamente asimilan la intercambiabilidad de ambos conceptos. El presente principio enfatiza la necesidad de que la información local a un módulo posea un nivel de protección de acceso (privacidad) por defecto, declarando como perteneciente al interfaz de acceso público únicamente la información estrictamente necesaria. Lo que aquí se enfatiza es, entre otras cosas, la clara separación entre interfaz e implementación, a la par que se refuerza la idea de acceso cuidadoso y selectivo a una información que de otra manera pudiera malusarse o corromperse. Una buena figuración didáctica la constituiría aquí la del tonel de vino, convenientemente cerrado para asegurar su conservación y únicamente accesible a través de un pequeño interfaz: el grifo.

Módulos Abierto-Cerrados: Según el enfoque clásico de diseño, un módulo cerrado es aquél que, superadas las etapas de testeo y validación de estabilidad, queda listo para su uso, a través de su interfaz, por otros módulos. Un módulo abierto sigue, en contrapartida, sujeto a posibles extensiones. ¿Qué ocurre, empero, cuando hay que modificar un módulo cerrado? Pues que éste debe ser abierto, modificado y sometido de nuevo al proceso de testeo y estabilización de uso, amén de que se genera la necesidad de actualizar los sistemas clientes del antiguo módulo. La solución de un OOSS es la siguiente: los módulos han de ser, a la vez, abiertos y cerrados. Si se detecta un problema en un módulo, o si éste debe ser modificado para cambiar algún aspecto de su comportamiento, en vez de

abrir y acceder directamente al módulo dado, de alguna manera éste se "clona" incorporándose, como parte física o conceptual, en otro nuevo, posiblemente superconjunto de aquél, de forma que es en este nuevo módulo donde se realizan los cambios adecuados. No hay necesidad, pues, de modificar los módulos existentes, de manera que no se pueden extender al sistema los posibles errores posiblemente cometidos en este proceso. Naturalmente se inserta uno o más nuevos módulos en el sistema, pero de acuerdo con el criterio de continuidad -y haciendo uso, normalmente, de la cualidad de polimorfismo que caracteriza a un OOSS-, esta pequeña adición originará un cambio pequeño o nulo en el sistema total. Precisamente la programación visual, básica y tradicionalmente sustentada en la interacción de módulos cerrados de utilización inmediata como "cajas negras", al aplicar extensiones de orientación-a-objetos empieza a posibilitar, en aplicación de las obvias ventajas de este principio, la "derivación visual" modular.

BOTTOM-UP VERSUS TOP-DOWN

El enfoque top-down se fundamenta en el refinamiento gradual de lo que se considera función principal o abstracta del sistema. De aquí fácilmente se infiere que, según este enfoque, todo sistema ha de poder ser sintetizado en una función "top", pero lo cierto es que los sistemas reales no poseen tal abstracción "top". El método top-down simplemente pregunta: ¿Qué debe hacer el sistema?, apoyándose así en la parte más volátil del mismo, el interfaz externo, y estableciendo prematuramente la secuenciación de las acciones a realizar. Una aproximación metodológica esencialmente más apropiada sería la basada en la sustancia del sistema: los datos. Pero no se trata aquí de una mera traslación del foco del esfuerzo de proceso de software desde las funciones a los datos, obteniendo poco más de una nueva problemática de características opuestas: en los OOSS's, por el contrario, las funciones encuentran, según Meyer, su exacto emplazamiento en los ADT's.

Lo más importante de la nueva orientación se resume en la premisa: primero hay que mirar a los datos, obviando el propósito efectivo del sistema, de tal manera que cuanto más tarde modelemos "lo que hace" éste, mejor. Se introduce así una esquematización denominada de "shopping list", donde se describen las operaciones susceptibles de ser aplicadas a cada módulo liberadas de las restricciones de orden. La pérdida, pues, de la prevalencia inicial de la secuenciación conlleva una aproximación al punto de vista del sistema a ser modelado, de manera que se generan sistemas robustos, no dependientes de la impulsión de las relaciones temporales, que pueden cambiar sin afectar a la estructura de aquellos. Se dice, así, que la construcción de un OOSS se realiza de abajo hacia arriba (bottom-up), en clara inversión del tradicional top-down. La realidad, sin embargo, es que la flexibilidad propia de los procesos de OOA y OOD origina una frecuente mixtura iterativa entre ambas técnicas, aunque, eso sí, supeditada a lo que se denomina enfoque "model-driven" o "responsability-driven".

EJEMPLOS DE CONCEPTUALIZACIÓN OBJETUAL

"Un objeto tiene un conjunto de operaciones y un estado que memoriza el efecto de las operaciones" (J. Bézivin)

"Un objeto es algo que existe en el tiempo y en el espacio y que puede ser afectado por la actividad de otros objetos" (Grady Booch)

"Un objeto es un concepto, abstracción o cosa con límites bien definidos y significado para el problema abordado" (James Rumbaugh et al.)

"Un objeto posee estado, comportamiento e identidad; la estructura y comportamiento de objetos similares están definidas en su clase común" (Grady Booch)

"Un objeto posee operaciones que definen su comportamiento y variables que definen su estado entre llamadas a las operaciones" (P. Wegner)

CICLO DE VIDA DE UN OOSS

El ciclo de vida de desarrollo software convencional está fuertemente focalizado en fases, constituyendo lo que se denomina ciclo en cascada (waterfall), de forma que cada fase utiliza diferentes técnicas y produce distintos resultados agrupados en su conclusión, que serán utilizados por otras fases subsiguientes. El ciclo de vida orientado-a-objetos sigue, en contrapartida, un esquema paralelo-recursivo en el que las tradicionales fases se solapan en el tiempo, debido a un desplazamiento de la focalización hacia los objetos, produciendo resultados discretos a lo largo de todo el ciclo, en un proceso involutivo de continuo refinamiento. Realmente, desde un punto de vista de gestión de proyectos, el énfasis en períodos temporales sí recae secuencialmente en fases concretas, pues, por ejemplo, el esfuerzo inicial se dirige sobremanera al análisis de requerimientos, decreciendo paulatinamente conforme el proyecto avanza. Así, por ejemplo, [Booch91] establece los siguientes porcentajes: Análisis (25%), Diseño (38%), Codificación (13%), Chequeo (19%) e Integración (7%).

La naturaleza recursiva del ciclo de vida de un OOSS queda patente, por ejemplo, en la síntesis de [Berar92]: "Analiza un poco, diseña un poco, implementa un poco, chequea un poco", así como en el modelo de fuente de [Hende90], de grafismo autoexplicativo. James Rumbaugh explica, de forma ingeniosa, que un desarrollo realístico de software es más como una piscina que como una cascada, esencialmente con agua en común.

TRANSICIÓN DEL OOA AL DISEÑO ORIENTADO-A-OBJETOS

James J.Odell ha afirmado que el OOA "no debiera modelar la realidad, sino más bien la forma en que la realidad es comprendida por la gente". Pero, ¿qué se entiende aquí por "modelo"? Según Michael Blaha, "una abstracción de algo con el propósito de comprenderlo antes de construirlo. Los modelos incluyen sólo aquellos aspectos relevantes a la solución de un problema; los detalles extraños son ignorados".

En la etapa de análisis se identifican entidades conceptuales, con contornos bien definidos, correspondientes a la abstracción de singularidades del escenario concreto por el que se matiza nuestro conocimiento del mundo real. En la etapa de diseño, seguidamente, tales entidades abstractas se trocan -directamente, en correspondencia biunívoca- en clases y objetos con las mismas estructura y organización. Se produce así, como vemos, un mapeo uno-a-uno entre componentes de las fases de análisis y diseño orientado-a-objetos, que algunos autores reclaman debiera ser extensivo a la etapa de implementación también (usando posiblemente de la misma notación y esquemas), y que aligera notablemente el tradicional "gap" entre SA/SD (Structured Analysis/Structured Design), suavizando hasta tal punto la transición entre OOA y OOD que se torna difícil establecer los líndes entre ambas fases. Naturalmente esto se debe al alto nivel de integridad conceptual y consistencia procurado por las metodologías de orientación--a-objetos.

En [Wasse91a] se establece que un auténtico OOA habría de identificar un conjunto de clases y las relaciones entre éstas, incluyendo las de uso y herencia, modelando así el conocimiento del sistema de una forma completa e inambigua, permitiendo que ese modelo pueda ser continuamente refinado y comunicado a otros. El hecho de que la fase de OOA no tenga por qué identificar todos los objetos de un sistema (pues durante el proceso de OOD serán identificados algunos objetos adicionales, a la vez que serán rechazadas algunas de las entidades halladas en el proceso de análisis), refuerza la naturaleza involutiva del ciclo de vida de un OOSS.

De cualquier forma, y a pesar del borroso linde entre ambas fases, es conveniente recordar que el Análisis es una actividad focalizada-en-el-problema (el qué), mientras que el Diseño está focalizado-en-la-solución (el cómo), de manera que es posible que en alguna ocasión la mera traslación directa a la fase de diseño de los resultados del análisis constituya una error de imprecisión, aunque sea cierto, en cualquier caso, que el sustrato conceptual trasladado ha de ser básicamente correcto.

DISEÑO ORIENTADO-A-OBJETOS

Según afirma Meyer, el OOD representa "la construcción de sistemas software como colecciones estructuradas de implementaciones de Tipos de Datos Abstractos" [Meyer88]. A fin de intentar extraer el sustrato común bajo los métodos de OOD, examinaremos tres de ellos:

I.. Método de GRADY BOOCH
- A.. Modelos de OOD
  - **¡Error! Argumento de modificador desconocido.**.. Lógicos
    - a.. Estructuras de clases
    - b.. Estructuras de objetos
  - **¡Error! Argumento de modificador desconocido.**.. Físicos
    - a.. Arquitecturas modulares
    - b.. Arquitecturas de procesos
- B.. Proceso de OOD
  - **¡Error! Argumento de modificador desconocido.**.. Identificación de clases objetos a un nivel dado
  - **¡Error! Argumento de modificador desconocido.**.. Identificación de relaciones entre clases y objetos
  - **¡Error! Argumento de modificador desconocido.**.. Implementación de clases y objetos
- C.. Diagramas de clases
  - **¡Error! Argumento de modificador desconocido.**.. Estructuras
  - **¡Error! Argumento de modificador desconocido.**.. Especificaciones
  - **¡Error! Argumento de modificador desconocido.**.. Relaciones

II.. Método de BERARD
- A.. Establecimiento del problema
- B.. Identificación de objetos candidatos
  - **¡Error! Argumento de modificador desconocido.**.. Identificación de objetos de interés
  - **¡Error! Argumento de modificador desconocido.**.. Asociación de atributos con objetos de interés
- C.. Identificación de operaciones (server/client) de objetos
- D.. Aplicación recursiva de OOD
- E.. Selección, creación y verificación de objetos
- F.. Decisiónes de implementación de objetos
- G.. Creación de modelos gráficos orientados-a-objetos
- H.. Establecimiento de los interfaces de los objetos
- I.. Implementación de los objetos

III.. Método de JAMES RUMBAUGH et al.
- A.. Modelado de objetos

Como bien se puede notar en las fases expuestas, una demasiado simple esquematización intuitiva de todas ellas podría resultar en la siguiente secuencia involutiva: descripción textual del problema, identificación y descripción de objetos, establecimiento y descripción de relaciones y comunalidades entre objetos, y descripción de los procesos temporales de cambios de estados de objetos.

CLASIFICACIÓN DE LOS MÉTODOS DE OOD

Dado que no existe en la actualidad un único método estándar de OOD, y ni siquiera hay convenio en la aceptación formal de los distintos conceptos básicos que conforman el paradigma de orientación a objetos, hablar de esquemas clasificatorios o evaluadores de las técnicas de OOD supone entrar en un segmento metodológico de escasa madurez formal, de manera que perfectamente cabría evaluar los métodos de evaluación a considerar, y así sucesivamente. En [Arnol91], por ejemplo, se decide la aplicación de criterios (soportes conceptuales, herencia, visibilidad, tiempo de vida, concurrencia, comunicación, clases de modelos, notaciones, contexto de desarrollo, cobertura de ciclo de vida, propiedades del proceso, recursos, accesibilidad y aplicabilidad) para finalmente concluir, obviando los típicos cuadros de evaluaciones parciales, con apreciaciones intuitivas de carácter genérico del tipo "el método Wirfs-Brock soporta totalmente los conceptos de orientación-a-objetos. El proceso es exploratorio e informal y resulta más apropiado para el desarrollador individual que para grandes equipos". [Berar92a], por otro lado, desarrolla una secuenciación evaluadora intentando evitar los defectos conceptuales en que afirma suelen recaer este tipo de trabajos, aunque resulta ciertamente chocante, a pesar de la seriedad del informe, la prevalencia que se le da al método de Booch y, sobre todo, al de Berard, autores del trabajo, basado grandemente en aquél. Las conclusiones de Berard sobre los distintos métodos se agrupan, sin embargo, en razón de las necesidades y posibles requerimientos de los potenciales usuarios de los mismos, por lo que puede resultar de utilidad práctica a la hora de afrontar una primera decisión electiva. La abundancia de estándares fuerza a que los informes comparativos confluyan en recomendaciones de uso, frecuentemente influidas por el bagaje subjetivo de los autores de los mismos.

Una clasificación genérica de los métodos de OOD sería cualificarlos como unarios o ternarios, dependiendo de si se basan en un enfoque globalizador desde el punto de vista de la orientación-a-objetos o más bien constituyen una adaptación de las tradicionales fases del proceso de SA/SD. Así, verbigracia, serían unarios los métodos de Coad/Yourdon y de Wirfs-Brock, mientras que resultarían ternarios los de Rumbaugh y Shlaer & Mellor. De esta manera los métodos ternarios aparecen, en un primer esbozo, especialmente indicados para procesos corporativos de transición al nuevo paradigma de objetos, mientras que los unarios resultan adecuados para los desarrolladores individuales y los procesos de asunción integral de la nueva orientación modular.

Un tercer estadio clasificatorio lo constituirían los métodos de OOA & OOD que pretenden una fusión de los esquemas estructurados con los de orientación-a-objetos, tales como "Synthesis" de Page-Jones & Weiss o el "Object-Oriented Structured Design (OOSD)" de Wasserman.

DIAGRAMANÍA

Resulta ciertamente descorazonadora la profusión de tantos elementos diagramáticos, escasamente diferenciados entre sí, provenientes de los esquemas gráficos de SA/SD. Ocurre, así, que al unificar la orientación--a-objetos las notaciones en el proceso entero de desarrollo, los autores de los distintos métodos, asumiendo la ventaja competitiva comercial que la extensión del uso de éstos efectivamente representa, los intentan particularizar de forma forzosamente matizada: donde en un método aparece un círculo, en otros aparece una flecha o un cuadrado; donde un cuadradado, en otros un círculo; etc.

Peter Coad directamente afirma: "Los tres ingredientes más importantes de cualquier proyecto son: gente, gente, gente. Quien desarrolla el trabajo es más importante que los métodos (OOA, OOD) o las herramientas (CASE, OOPLs)", así que recomienda "reclutar gente de calidad ... y después proveer métodos y herramientas" [Coad91].

Dada la importancia que en todos los métodos se le concede a la identificación de clases (u objetos, en un sentido más amplio), las especificaciones textuales propias de la etapa de OORA (Object-Oriented Requirements Analysis) aparecen especialmente relevantes. De hecho, todos los métodos de OOA & OOD parten de tales especificaciones, de donde se desprende que una adecuada representación textual, quizás ayudada por un simple soporte informal de comprensión gráfica, podría resultar autosuficiente para la gestión de las fases de OOA & OOD. Tanto es así, que parece se está recuperando una cierta neurosis textual que, a la vez que autodocumenta el proceso de desarrollo software, recoge la insatisfacción generada por la artificial practicidad de las actuales herramientas gráficas. Quizá el método más cercano a esta informalidad gráfica sea el de Wirfs-Brock: exploratorio, antropomórfico y basado en descripciones textuales traspasadas a fichas CRC.

MODELADO DE JERARQUÍAS

Una parte importante e intersección común de las técnicas de OOA y OOD es la disposición en jerarquías de herencia de las entidades modulares ya identificadas. Tal y como se ha notado al principio de este trabajo, el autor estima que la herencia es más un mecanismo que una base conceptual, pero hay que tener en cuenta que las técnicas de OOD (como en su día las de SD) provienen de la abstracción y generalización de los comportamientos observados en la OOP (eppore il muove...), de forma que si en la inmensa mayoría de los OOPL's se contempla tal mecanismo, parece claro que las técnicas de OOD deberán también soportarlo en clara medida.

Las facilidades de reutilización modular procuradas por la herencia han ocasionado lo que Cargill denomina "Problema de la herencia innecesaria" y

que Rumbaugh simplemente tacha de "Abuso de la herencia". Es desafortunadamente frecuente que, en las etapa de OOD, las relaciones entre clases se modelen, sin restricción ni control alguno, como derivativas por herencia, lo que suele conducir a situaciones esencialmente erróneas, artificiosas e, incluso, peligrosas para la futura reutilización de los resultados del diseño.

Naturalmente, las técnicas de OOA/OOD tienden a procurar guias para el correcto modelado de una jerarquía de este orden, en general de acuerdo con el siguiente criterio: una relación de herencia únicamente estará verdaderamente cualificada cuando se produzca una especialización o refinamiento mediante la adición en la subclase de comportamientos y/o atributos distintos a los de la superclase de que aquélla deriva. Lo que, en definitiva, se pretende instruir podríamos sintetizarlo en la frase: "Una relación de herencia necesariamente ha de ser una relación ES-UN". Pero aún esto podría, en ciertos ámbitos, resultar confuso: de hecho, no todas las relaciones del tipo ES-UN han de modelarse como herencia, (y aquí entraríamos en colisión con el mecanismo de *layering*, perfecta disyuntiva de la herencia a los solos efectos de subencapsulación modular semántica).

Uno de los errores más comunes en el modelado de jerarquías se da al considerar como herencia el refinamiento por restricción. De esta manera aparecen normalmente en textos introductorios a la OOP ejemplos erróneos del tipo: un cuadrado deriva (por hereda) de un rectángulo, o una circunferencia deriva de una elipse. Piénsese, verbigracia, que una circunferencia NO ES-UNA elipse, por mucho que buena parte del código de esta última clase pueda ser empleado en aquélla: una cancelación o modificación, en determinados entornos, de la condición restrictiva podría evidenciar el desajuste de comportamiento entre ambas clases. Así ocurre, por ejemplo, con el redimensionamiento de figuras en un editor gráfico interactivo: si una circunferencia mantiene el comportamiento de una elipse, la variación de su tamaño originará por fuerza una elipse, al haber desaparecido, merced a la propia operatividad del editor, la restricción impuesta sobre la igualdad de los dos radios en la elipse: pero ¡un objeto no puede cambiar de clase!. Un ejemplo más claro, en el mismo entorno, lo podría dar la relación entre cuadrado y rectángulo, que, en ortodoxia conceptual, se limitaría a compartir una misma superclase.

Con el fin de proporcionar una guia, la aplicación inseparable de los tres siguientes criterios ayudará sobremanera a establecer la correctitud de un modelo jerárquico:

> *La relación entre dos clases será modelada como herencia si y solo si se cumple que:*
> *1) la superclase mantiene una relación de inclusión estricta con respecto a la de su subclase.*
> *2) donde quiera que aparezca un objeto de la subclase se puede aplicar un objeto de la superclase.*

3) es necesario posibilitar la actuación polimórfica de los objetos de las clases en relación.

La consideración de la primera condición pudiera obligarnos a reconsiderar una relación entre dos clases y abstraer las partes comunes en una nueva superclase de ambas. El uso combinado de gráficos derivativos y de diagramas de Venn, tal y como propugna [Wirfs90], ayuda grandemente a su comprobación.

La segunda condición asegura la filtración de relaciones ES-COMO-UN, ES-PARTE-DE, etc., a la vez que evita la herencia errónea por agregación o restricción.

La última opción establece, por último, que, aun correspondiendo la cualificación de herencia a una relación, ésta no será aplicada si no se pretende tomar ventaja de tal mecanismo, pudiendo sustituirlo por otros más sencillos, como el de layering.

Adicionalmente cabría decir que en absoluto hay que minorar la importancia de la herencia en un OOSS. James Rumbaugh ha llegado a afirmar que "un modelo orientado-a-objetos es orientado-a-objetos porque el potencial de añadir herencia al modelo está siempre presente". De hecho, en ciertos autores, la jerarquización de reuso es denominada herencia sin restricciones (unrestricted inheritance).

Una clarificadora sustanciación del enfoque correcto del modelado de la herencia aparece en [Bar-D92], que asimila las muchas definiciones de herencia en dos categorías: herencia de comportamiento (semántica) y herencia de implementación, para concluir que las jerarquías deben adscribirse a la primera.

DISEÑO ORIENTADO-A-OBJETOS DE BASES DE DATOS

¿Qué ocurre, desde la óptica de la orientación-a-objetos, con el esfuerzo tradicionalmente focalizado en el análisis y diseño de bases de datos relacionales adscritas al problema a modelar? La respuesta es sorprendentemente simple: **NO** hay análisis ni diseño sustantivo que aplicar a los OODBMS's (Object-Oriented DataBase Manager Systems). Digamos que, en una aproximación peligrosamente intuitiva, los objetos sabrán cómo archivarse y recuperarse en el espacio de una aplicación: el gestor de la base de datos tomará cuenta de estas y otras operaciones, liberando al desarrollador de su modelado expreso. Naturalmente la revisión de las bases de objetos (o con extensiones a objetos) exceden el ámbito de este trabajo, pero debido a su importancia merecen, al menos, una sucinta y elemental descripción extensiva.

Los OODBMS's añaden a las facilidades del modelo relacional (modelo de datos, persistencia, concurrencia, gestión de transacciones, recuperación, lenguaje de consulta, performance y seguridad) otras nuevas (abstracción de datos, potentes capacidades de modelado de información, identidad objetual, encapsulación y ocultación de la información, datos activos, herencia, funciones y datos polimórficos, composición, paso de mensajes y extensibilidad). En [Butte91] se resumen, por otra parte, las nuevas y potentes capacidades transaccionales de los OODBMS: grandes transacciones, transacciones anidadas, control optimizado de concurrencia, concurrencia híbrida y versioning.

Tras lo expuesto, cabría preguntarse: ¿verdaderamente el modelo orientado-a-objetos sustituirá al relacional en gestión de bases de datos? Parece que sí: la compañía londinense de consultoría OVUM afirma en un informe muy extendido titulado "Database for objects: the market Oportunity" que las bases de datos relacionales con extensiones a objetos llegarán a suponer el 52% del mercado total de DBMS's en 1.995, con un mercado calculado en 4 billones de dólares entre USA y Europa. OVUM estima, así mismo, que el mercado de las bases de objetos puras alcanzarán los 560 millones de dólares en el mismo año.

ALGUNAS HERRAMIENTAS OO-UPPER-CASE Y OO-I-CASE

La siguiente relación no pretende ser exhaustiva, sino únicamente evidenciar las tendencias de mercado en el soporte de las distintas metodologías de OOA & OOD. Se ha obviado, de cualquier forma, la descripción del soporte de SA/SD por varias de las herramientas expuestas.

Herramienta	Compañía	Métodos soportados
Objectory	Objective Systems	Objectory
System Architect	Popkin Soft & Sys	Booch, Coad/Yourdon
Teamwork OOA	Cadre Technologies	Shlaer-Mellor, HOOD
Teamwork OOD	Cadre Technologies	HOOD
OOATool	Object International	Coad/Yourdon
OODTool	Object International	Coad/Yourdon
Rational ROSE	Rational Inc	Booch
ILOG Kads Tool	ILOG	Common Kads
SES/Objectbench	Scientif.& Eng.Software	Shlaer-Mellor
001 Tool Suite	Hamilton Technology	Dev.Before the Fact
ObjectCraft	ObjectCraft Inc	ObjectCraft
Ipsys ToolBuilder	Ipsys Ltd.	HOOD
LOV/Object Edit.	Verilog	Rumbaugh
ObjecTime	ObjecTime Ltd.	ROOM
ObjectMaker	Mark V Systems	Booch, Coad/Yourdon, Rumbaugh, Wirfs-Brock, Shlaer-Mellor, ...

Objecteering	SofTeam	Class Relation
OOSD/C++	IDE	OO Structured Design
Stood	TNI	HOOD
Object M.B.	IntelliCorp	Martin & Odell OO IE
Paradigm Plus	Protosoft Inc.	Booch, Coad/Yourdon, Rumbaugh, HOOD, EVB, Fusion
OMTool	General Electric	Rumbaugh
Iconix P.T.	Iconix Soft Eng.	Rumbaugh, Booch, Coad/Yourdon
Object Sys/Desig	Palladio Software Inc	Booch
OOther	Roman Zielinksi	Coad/Yourdon
TurboCase	StructSoft	Wirfs-Brock
ATRIOM	Semaphore	Booch, Coad/Yourdon, Rumbaugh, Shlaer-Mellor
Foundation	Andersen Consulting	Foundation

REFERENCIAS

[Arnol91] Patrick Arnold, Stephanie Bodoff, Derek Coleman, Helena Gilchrist & Fiona Hayes, "An evaluation of five object-oriented development methods", Hewlett Packard Laboratories, HPL-91-52, 1991.

[Bar-D92] Tsvi Bar-David, "Practical consequences of formal definitions of inheritance", JOOP, 5(4), 1992.

[Berar92] Berard Software Engineering Inc., "A Project Management Handbook for Object-Oriented Software Development, Volume 1", Berard Software Eng Inc, 1992.

[Berar92a] Berard Software Engineering Inc., "A Comparison of Object-Oriented Development Methodologies", Berard Software Eng Inc, 1992.

[Booch91] Grady Booch, "Object-Oriented Design with Applications", Benjamin/Cummings, 1991.

[Butte91] Paul Butterworth, "ODBMS as database managers", JOOP, 4(2), 47-50, 1991.

[Coad91] Peter Coad, "Why use object-oriented development (A management perspective)", JOOP, 4(6), 1992.

[Hende90] Brian Henderson-Sellers, "A Book of Object-Oriented Knowledge", 1990, Prentice Hall.

[Meyer88] Bertrand Meyer, "Object-Oriented Software Construction", 1988, Prentice Hall.

[Rumba91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy & William Lorensen, "Object-Oriented Modeling and Design", 1991, Prentice Hall.

[Wasse91] Anthony I. Wasserman, "Object-Oriented software development: issues in reuse", JOOP, 4(2), 55-57, 1991.

[Wasse91a] Anthony I. Wasserman, "From Object Oriented Analysis to Design", JOOP, 4(5), 1991.

[Wirfs90] Rebeca Wirfs-Brock, Brian Wilkerson & Lauren Wiener, "Designing Object-Oriented Software", 1990, Prentice Hall.

A-4

INSISTENTE PERSISTENCIA

La *persistencia* se ha convertido en la palabra mágica alrededor de la que se han aglutinado sistemas, diseños, expectativas y, finalmente, un cierto desencanto. Ante la lectura de algunas publicaciones técnicas podría afirmarse, parafraseando a Swift, que "*es tan difícil librarse de la persistencia como del infierno*". El no buscado hermetismo conceptual habitualmente aplicable a los sistemas de *objetos* se ha trocado en opaca y esponjosa tibiedad envolviendo al concepto de *persistencia* y convirtiéndolo en borrosa zona de convergencia de distintas metodologías para el archivo y acceso de datos, obviando la conceptualización teórica común a los distintos esquemas prácticos implementados. El concepto se ha modelado, en definitiva, por inferencia implícita basada en los desarrollos prácticos "propietarios" de distintas empresas y entidades, adoleciendo de un soporte axiomático propio.

Encontramos una brevísima aproximación a la persistencia como cualidad general de los objetos en [Meyer88], aunque en este texto se resolvía su discusión asimilando tal cualidad a las propiedades implementadas en **Eiffel** mediante la clase **Storable**. El Dr. Meyer incidía puntualmente sobre el interesante problema de la recuperación a la memoria de objetos cuyas clases han cambiado desde el momento de su archivo. Por último, y teniendo en cuenta que en aquel momento empezaban a despuntar los albores de la futura gran eclosión de ODBMSs (Sistemas Gestores de Bases de Datos de Objetos, también denominados OMSs: Sistemas Gestores de Objetos) , estimaba improbable que las técnicas de extensión de la persistencia a objetos pudieran mejorar la efectividad de los esquemas de bases de datos tradicionales (relacionales, jerárquicas, de red, de relación-entidad). Las últimas pruebas realizadas sobre ODBMSs han arrojado, empero, una relación de efectividad en archivo y recuperación de objetos comprendida entre 36:1 y 100:1 con respecto a los procedimientos homólogos sobre equiparables estructuras de datos tradicionales basados en RDBMSs (Sistemas Gestores de Bases de Datos Relacionales). Tales resultados han situado la *persistencia* en un contexto más atractivo, enfatizando el desarrollo de distintas aproximaciones al concepto.

La persistencia es todavía, no obstante, "la gran desconocida": en un texto de OOA&OOD importante cual es [Rumba90] el tratamiento que se da a esta propiedad es poco menos que trivial y, aún así, limitado a escasas líneas, y no muy inferiores en número a las dedicadas a los que en el texto se denominan OO-DBMSs y que se asimilan como heterodoxa unión de un OOPL y de persistencia *de datos*.

En el presente anexo abordaremos una propuesta de formalización teórica de la persistencia en subsistemas de objetos. Repasaremos, seguidamente, uno de los esquemas de persistencia más extendidos, comentando en detalle la implementación adoptada por Borland en sus librerías-entornos de clases para Borland C++ 3.1. Examinaremos, por último, un método automático, basado en el procedimiento anterior, para añadir persistencia a las clases definidas en un sistema C++.

FUNDAMENTOS CONCEPTUALES

La Guía de Arquitectura de Gestión de Objetos confeccionada por el OMG (*Object Management Group*) establece el siguiente tenor literal:

> *"Los objetos persistentes se utilizan para representar el estado a largo plazo de una computación en ejecución, como por ejemplo el conjunto de empleados de una corporación y las relaciones entre los mismos. Tales instancias poseen, por ejemplo, la cualidad intrínseca de sobrevivir al proceso que las creó; incluso sobreviven a la extinción temporal de la CPU que las generó.*

Se identifica aquí, pues, la persistencia con la cualidad de algunos objetos de mantener su identidad y relaciones con otros objetos con independencia del sistema o proceso que los creó. Tal propiedad supone, por ejemplo, la congelación temporaria de un determinado proceso inter-objetos (pensemos en una gran transacción a realizar por un ODBMS).

El concepto de persistencia aparece intuitivamente ligado al de almacenamiento, sosteniendo únicamente como necesario corolario al proceso de recuperación. En realidad ambos procesos forman parte de la cualidad definida como persistencia. En adelante, pues, hablaremos de persistencia, o impulsión de persistencia (concepto fuertemente ligado a los sistemas persistentes dinámicos), más que de archivo o recuperación de objetos. El sistema determinará en cada momento, de forma transparente para el usuario en el caso más general, si debe usar el almacenamiento persistente secundario o primario.

La persistencia de un sistema de objetos se implementa, en la práctica, proveyendo a éstos de un mecanismo cuyo objetivo básico consiste tanto en el almacenamiento de objetos existentes en la memoria como en la

recuperación posterior a ésta de los mismos. La eficiencia es un factor crítico fuertemente ligado a tal mecanismo, que supone la preservación no sólo de los datos estrictamente contenidos en un objeto (de tipo incorporado e instancias de otras clases) sino también de la identidad del mismo (determinada tanto por las características semánticas de tipo adquiridas mediante herencia -derivación en C++- como por las relaciones contextuales con respecto a funciones con ligadura dinámica -tabla de funciones miembro virtuales en C++-), así como de las relaciones, apuntadores y referencias a otros objetos. Definiremos a efectos prácticos la ***persistencia***, pues, como ***la cualidad de un determinado objeto de almacenar y recuperar eficiente, automática y selectivamente la estructura compleja de objetos relacionados con el mismo (incluimos aquí, naturalmente, la auto-relación)***. De acuerdo con tal definición la aplicación de la persistencia a un objeto inmerso en una estructura equivaldría a la impulsión de un primer mensaje al mismo que, a su vez y en razón de un mecanismo de extensibilidad que veremos más adelante, posiblemente generaría otros mensajes dirigidos a sendos objetos de la estructura y daría lugar al trazado figurado de un complejo grafo derivativo en la estructura en sí, transformándose así en un mensaje que recorrería la estructura de objetos procediendo bien al almacenamiento bien a la recuperación en memoria de los objetos calificados como persistentes.

Como vemos, la persistencia en un sólo objeto constituiría únicamente un caso particular de restricción selectiva de la extensibilidad de la más general cualidad de persistencia aplicable a una estructura de objetos: o bien el objeto en sí no contiene ni apunta o referencia a otros objetos, o bien implícita o expresamente se han declarado tales objetos, directamente o a través de referencias o punteros, como **transitorios** (*transient objects*). Así observamos que la persistencia modela distintos grados de extensibilidad inter-objetos que podrían ser finamente granulados, permitiendo así una gran flexibilidad en la implantación de la misma.

Por extensión, la cualidad de persistencia habrá de aplicarse, pues, a la estructura en sí como caso más general. Intentaremos establecer seguidamente, así, el soporte teórico básico de la persistencia de sistemas de objetos.

IMPULSIÓN DE PERSISTENCIA

Examinemos en primer lugar la cualificación de la impulsión inicial de persistencia: ¿Debe ser ésta un mensaje dirigido a un objeto de la estructura, que contaría con un método recursivo propio o heredado para su aplicación extensiva? ¿O más bien debe constituirse en un mensaje que el objeto al que ha de aplicarse la impulsión enviará a otro objeto, que denominaremos *manipulador de persistencia*, soportando éste las características concretas de la persistencia deseada?

Evidentemente el primer método ligaría de forma indeleble las características físicas del sistema de persistencia elegido a la definición de las clases implicadas, vulnerando así uno de los principios básicos de la OOP: cambios pequeños han de tener pequeñas repercusiones. En un tal sistema un pequeño cambio en el sistema de archivo y recuperación de objetos implicaría una modificación del código tanto más importante cuanto más extenso sea el conjunto de objetos afectados. La extremada concrección que podría derivarse del conocimiento profundo de la estructura interna de una clase podría, por otra parte, resultar en una demasiado ajustada implementación de la persistencia, que a su vez podría redundar en detrimento de la reutilización de tal código. Hay que considerar, no obstante, la posibilidad que cada código de esta forma implantado revierta en un objeto o estructura de objetos exterior en calidad de *gestor de persistencia*, de forma que esta primera posibilidad se convierta en un caso especial de la segunda opción. En general éste es el procedimiento usado cuando la persistencia se direcciona siempre a través del mismo gestor o *persistence manager*, cual es el caso habitual en los ODBMSs.

El segundo procedimiento usaría de *manipuladores persistentes*: objetos autónomos o insertos en una estructura gestora de persistencia. A un nivel elemental, éste es el trabajo realizado, por ejemplo, por los objetos *ofstream* e *ifstream* cuando se les remiten los mensajes de extracción (<<) e inserción (>>) respectivamente:

```
ofstream opo("archivo.001", ios::out );
// inserta un objeto en un objeto ofstream
opo << objetoDeTipoX;
ifstream ipo("archivo.002", ios::in );
// extrae un objeto de un objeto ifstream
ipo >> objetoDeTipoY;
```

Este método exige que los objetos gestores de persistencia ***pmg****s* (*persistence managers*) sepan cómo tratar a cada objeto. De acuerdo con el anterior código esto supondría que en la descripción de las clases de los respectivos *pmgs* aparecieran unas funciones definidas posteriormente como las siguientes:

```
ofstream& ofstream::operator <<( const TipoX& x ) { /* ... */ }
ifstream& ifstream::operator >>( const TipoY& y ) { /* ... */ }
```

o posiblemente también como

```
inline ifstream& operator >>( ifstream& iso, const TipoY& y )
{ /* ... */ }
```

En realidad, el hecho de tener que variar las clases de los *pmgs* cada vez que se deseara dotar de persistencia a un objeto (obviaremos aquí las facilidades de derivación) atenta contra el núcleo teórico de la OOP, generando una fácilmente monstruosa macro-clase virtualmente no-reutilizable. Así, en la

práctica, se direcciona el control de la persistencia hacia un método encapsulado en el mismo objeto inicial del subsistema que se desea persistente, el cual, a su vez, convenientemente revierte el control de nuevo al *pmg*, bien para cada objeto de la secuencia impulsora o bien sólo para los tipos incorporados, y siempre una vez finalizada la impulsión. De hecho estamos reiterando el primer esquema propuesto, pero con la posibilidad de cambiar en tiempo de ejecución el *pmg*. De cualquier forma esta característica también podría ser implementada en la primera opción añadiendo, por ejemplo, un nuevo parámetro a cada método de persistencia encapsulado en una clase que soportara el paso de objetos *pmg*.

La anterior revisión ha confluido en resaltar la igualdad del sustrato básico sobre el que se asientan las dos opciones de impulsión de persistencia. La elección de una u otra depende más de consideraciones de diseño y estilo que de diferencias metodológicas. Más adelante examinaremos distintas posibilidades prácticas.

SISTEMAS PERSISTENTES

Los objetos de un sistema interrelacional **COS** (Complex Object Structure) afectados a través de la aplicación de la persistencia en un primer objeto **psg** (persistent subsistem generator) (initial persistence object) constituirán un subsistema **POSS** (Persistent Object SubSistem) de la estructura general que calificaremos como *susbsistema persistente en **COS** por **psg*** y que notaremos como

persistent<COS:psg>POSS

Tomemos como ejemplo el siguiente conjunto "OBS" de objetos, de cuyas identidades haremos abstracción, constitutivo de una estructura compleja de relaciones mutuas:

OBS={objP1, obj2, objP3, objP4, obj5, obj6, objP7, obj8, objP9}

donde objN representa la instanciación de una determinada clase y objPN un objeto dotado de persistencia. Si dirigimos la impulsión de *persistencia* a, por ejemplo, el objeto objP4, éste distribuirá tal mensaje a través de la estructura (para simplificar obviaremos en este estadio las referencias cíclicas y la redundancia de objetos), causando el archivo persistente de, verbigratia, los objetos objP1, objP3 y objP9. Representaremos el subsistema afectado SOBS con la siguiente notación:

persistent<OBS:objP4>SOBS { objP1, objP3, objP4, objP9 }

y calificaremos a objP4 como un objeto **spg** generador del mismo. Denominaremos **orden** de un subsistema persistente, por otro lado, al número de objetos generadores del mismo, y lo notaremos como

o(persistent<COS:iob>POSS)

de forma que, evidentemente, si **n** es el número total de objetos del subsistema persistente, siempre se cumple que

```
1 <= o(persistent<COS:iob>POSS) <= n
```

En el caso que el orden de un subsistema sea igual al número de objetos en él contenidos (**o(POSS)=n**), todos los objetos del subsistema serían generadores del mismo, recibiendo éste el nombre de subsistema persistente cíclico **CPOSS** (Ciclyc Persistent Object SubSistem), siendo representado por la notación

```
persistent<OBS>CPOSS
```

Supuesto que el orden de persistencia del anterior subsistema sea mayor de uno y dados dos generadores objP4 y objP9 del mismo, siempre se cumplirá que

```
persistent<objP4:OBS>POSS == persistent<objP9:OBS>POSS
```

Por el contrario la representación física del almacenamiento del subsistema persistente dependerá, en su caso más general, del generador utilizado. Si la representación física de un subsistema generada por dos distintos objetos *psg* es la misma, tales objetos se denominarán **generadores conmutativos de persistencia** del sistema. El conjunto de los generadores conmutativos de un subsistema dado se denominará **núcleo del subsistema**. Cuando el conjunto de generadores coincida con su núcleo, al subsistema generado se le denominará **subsistema persistente**.

En realidad el sistema axiomático esbozado soportaría una cualidad general de persistencia caracterizada por las propiedades de *concurrencia*, *mantenibilidad*, *inspeccionabilidad*, *reutilizabilidad de código*, *almacenamiento multinivel* y *estructuración dinámica*. Como quiera que el somero análisis de tales singularidades sobrepasaría con mucho el ámbito y las posibilidades de este anexo, se remite al lector a [Devis94], donde se desarrolla en detalle esta propuesta de formalización axiomática de la persistencia.

LA PERSISTENCIA SEGÚN BORLAND

Borland ofrece en la versión profesional de su compilador Borland C++ 3.1 una particular implementación de la persistencia estática de objetos, basada en la librería NIH expuesta en [Gorle90] y apoyada sobre la librería estándar *iostream* de C++. En realidad Borland ofrece dos versiones de tal mecanismo de persistencia insertadas en sus entornos comerciales (más que librerías) *TurboVision for C++* y *ObjectWindows for C++* (ofrecido éste último para su estandarización por el OMG), aunque ambas comparten los mismos identificadores, estructuras y procedimientos (a excepción de algunos *typedef*s). El lector podrá encontrar en [Urloc91] una comprehensiva y sucinta presentación de los entornos-marco de Borland.

Examinaremos, pues, en lo que sigue, únicamente una de tales implementaciones: la referida a *ObjectWindows for C++*, pudiendo fácilmente los lectores mapear todo lo expuesto al entorno para DOS. Intentaremos ahondar, por fin, en las consideraciones de diseño y metodología subyacentes en la construcción de tal sistema comercial, favoreciendo, en lo posible, aproximaciones mejoradas al mismo.

El mecanismo de persistencia adoptado en *ObjectWindows* es **dual**, **estático**, y **derivativo**. Veamos tales características en detalle:

El mecanismo de persistencia es **dual**: los objetos y sus **DAG**s (Direct Acyclic Graph) asociados deben ser expresamente archivados desde la memoria al almacenamiento persistente secundario (ficheros persistentes o *streams*) y expresamente recuperados desde éste a la memoria de la aplicación, o almacenamiento persistente primario.

Decimos que el mecanismo es **estático** porque implica que la cualidad de persistencia no pueda ser añadida o cambiada a un objeto en tiempo de ejecución. Esto es, se trata de una operación sobre las clases y no sobre los objetos. Los subsistemas persistentes quedan establecidos, así, en tiempo de compilación por la pertenencia de los objetos a determinadas clases *persistentes*. La persistencia estática de objetos deriva, pues, en el concepto de persistencia de clases. De esta manera los **OID**s (*Object Identification*s) se trocan en **CID**s (*Class Identification*s), y los mecanismos de identificación en tiempo de ejecución de un objeto con respecto al subsistema persistente son establecidos en tiempo de compilación y comprenden información lógica únicamente.

El sistema persistente se califica como **derivativo** porque los métodos de archivo y recuperación de objetos se implementan en las clases a los que éstos pertenecen de forma que, para evitar una excesiva dependencia de los métodos físicos, se hacen derivar de una clase

De acuerdo con lo expuesto, si deseamos convertir en persistentes a <u>todos</u> los objetos de una determinada clase debemos, en primer lugar, expresamente añadir a su descripción la derivación pública de la clase *TStreamable*:

```
class MiClasePersistente : [MiClaseBase,]* public TStreamable
{ /* ... */ };
```

A la clase se le "adhieren" entonces, por derivación, las funciones miembro de *TStreamable*, definidas todas ellas como funciones virtuales puras bajo las cualificacionesde acceso *protected* y *private*:

```
protected:virtual void* read( ipstream& ) = 0;
private:virtual const char* streamableName() const = 0;
protected:virtual void write( opstream& ) = 0;
```

Lo que se pretende con este mecanismo es forzar la redefinición de tales funciones o su redeclaración como funciones virtuales puras en la clase derivada. Este comportamiento, ajustado a la versión AT&T C++ 2.1 cambia, sin embargo, en AT&T C++ 3.0 permitiendo la herencia por defecto de las funciones virtuales de clases base en clases derivadas, que serían así declaradas abstractas. Esta matización no supone, empero, cambio alguno en el mecanismo de persistencia, a no ser la ocasional desaparición del chequeo por error en compilación debido a la no redefinición de tales funciones virtuales.

Debemos, pues, retomando el procedimiento, redefinir tales funciones en la clase a la que queremos dotar de persistencia. Y tal redefinición se implementará en directa dependencia y consecuencia del método de recuperación de objetos elegido. En este caso la aproximación a la persistencia se ha significado mediante la descomposición de los objetos en entidades de tipo incorporado (*int, char*, etc.) delimitadas por identificadores de clase. El mecanismo de recuperación de objetos se limitaría a clonar objetos creados a partir de tales identificadores copiando recursivamente sus miembros: los de tipo incorporado serían copiados directamente, mientras que los objetos apuntados, contenidos o referenciados serían recuperados desde el *stream* por el mismo procedimiento. Supondremos en lo que sigue que las funciones virtuales de *TStreamable* han sido ya redeclaradas en nuestra clase *aspirante-a-persistente*, de forma que nos ocuparemos únicamente de su definición fuera del ámbito de descripción de la clase.

En primer lugar tenemos que proporcionar un identificador a la clase compartido por todos sus objetos y accesible en tiempo de ejecución: esto se consigue mediante la función

```
inline const char* MiClasePersistente:streamableName() const
{ return "MiClasePersistente"; }
```

que devuelve el nombre de la clase, declarada en la sección *private* de ésta. En los mecanismos de persistencia derivados de la librería NIH existen funciones parejas que, básicamente por razones de eficiencia, devuelven un número en lugar de una cadena. Una solución alternativa consistiría en añadir un nuevo miembro a la clase *aspirante-a-persistente*:

```
public: static const char* const name;
```

el cual sería accedido por medio de la función definida así:

```
class MiClasePersistente : public TStreamable {
private:
        virtual const char* streamableName() const {return name;}
};
```

supuesta la declaración, en algún lugar del código, de la asignación:

```
const char* const MiClasePersistente::name="MiClasePersistente";
```

La declaración en la sección *private* restringe selectivamente el uso de la función *streamableName()* a las clases *friend opstream* e *ipstream*.

LA FUNCIÓN ESCRITORA

El siguiente paso es definir, en la sección *public*, la función de almacenamiento o *escritora* (write()) sobre el soporte de dos formas de indirección recursiva: el método de almacenamiento de la porción del objeto heredada por derivación se desvía hacia las correspondientes clases base, mientras que el procedimiento de archivo de los objetos de tipo no-incorporado se direcciona hacia éstos, forzando la *persistencia* tanto de las clases base como de las correspondientes a objetos contenidos en la clase *aspirante-a-persistente*. El argumento de la función *escritora* es una referencia a un objeto de clase *opstream (output persistent stream)*, derivada de *pstream* para operaciones de inserción, donde *pstream* es la clase base de la versión estructural "persistente" de la librería *iostream*. Realmente la diferencia entre ambas librerías radica en el tratamiento que la clase *pstream* realiza con los datos antes de su direccionamiento al buffer *streambuf* definido en *iostream*. Veamos un molde de la función:

```
void MiClasePersistente::write( opstream& ops )
{
        MiClaseBase::write( ops );
        ops << objetosDeTipoIncorporado;
        ops << objetoDeOtraClasePersistente;
        ops << punteroAObjetoDeOtraClasePersistente;
}
```

Este planteamiento exige, pues, que la descripción de MiClaseBase haya sido convenientemente adaptada para soportar el mismo tipo de persistencia. De esta forma deberíamos declarar recursivamente

```
class MiClaseBase : MiOtraClaseBase,
                                    public TStreamable { /* ... */ };
class MiOtraClaseBase : OtraClaseBaseMas,
                                    public TStreamable { /* ... */ };
// etc., etc.
```

Tal mecanismo induce, sin embargo, la siguiente penalización en su aplicación a estructuras de clases interrelacionadas por derivación como, por ejemplo, los entornos de clases *Smalltalk-like*: la continua aplicación de la derivación en tales clases implica la insistente e incremental repetición de la porción correspondiente a la clase *TStreamable* en todas y cada una de las clases *candidatas-a-persistentes*, lo cual, dado el anterior código y suponiendo la derivación múltiple siguiente:

```
class MiClaseSospechosa: public MiClaseBase,
                         public MiOtraClaseBase { /* ... */ }
```

podría dar lugar a un error por ambigüedad en la llamada, por ejemplo, a una función que tome como argumento un objeto o referencia a un objeto

del tipo de esta clase. Consideremos una de las definiciones del operador de inserción:

```
opstream& operator <<( opstream& ops, TStreamable& t )
{ /* ... */ }
```

Podemos aislar el problema en el siguiente código:

```
MiClaseSospechosa objetoSospechoso;
// la próxima línea ocasionará un error por ambigüedad,
//  pues el objetoSospechoso posee al menos dos
//  porciones TStreamable: la obtenida por derivación
//  de MiClaseBase y la derivada de MiOtraClaseBase.
//  ¿Cuál emplear?
ops << objetoSospechoso;    // ERROR
// el siguiente cast  resolvería la ambigüedad
ops << (MiOtraClaseBase)objetoSospechoso;
```

Cabrían dos posibles soluciones: bien trocar cada derivación en derivación virtual, bien aplicar la derivación de *TStreamable* únicamente a la clase base del entorno derivativo. Dada la singularidad **estática** del mecanismo aquí estudiado, aparece más apropiada la segunda opción, aplicable en todo caso a la clase base de la que suelen derivarse en su estadio inicial de diseño las restantes clases de un proyecto: tal clase se constituiría, así, además de en raíz del mecanismo de depuración (*debug*) del sistema en la base de persistencia del mismo. Hay que notar que la declaración de una derivación dada de *TStreamable* como *virtual* ocasionaría una dificultad adicional: dado que el mecanismo de gestión de la persistencia está basado en las facilidades provistas por la clase *TStreamable*, es frecuente el *cast* operacional a tal clase de los objetos persistentes; la derivación virtual ocasiona, sin embargo, que la clase base afectada sea accedida por medio de un puntero, de manera que un *cast* a *TStreamable* equivaldría al recorrido del puntero en sentido inverso: esto supone, evidentemente, la inaccesibilidad directa a la porción de la clase base. La accesibilidad indirecta podría ser implementada, con todo, mediante un mecanismo que recorriera el esquema jerárquico y estableciera lo que se denominan *relacionamientos inversos*, en un esquema de correspondencia parecido al empleado en la librería NIH.

Naturalmente en estructuras complejas de derivación mediante herencia múltiple cabría la aplicación selectiva simultánea de las dos opciones. De esta manera, y supuestas las circunstancias descritas, el código anterior podría reescribirse de la siguiente forma:

```
class MiPrimeraClaseBase :
        [claseBaseNoPersistente,]* virtual public TStreamable
        { /* ... */ };
class MiSegundaClaseBase :
        [otraClaseBaseNoPersistente,]* public TStreamable
        { /* ... */ };
class MiClasePersistente :      public MiPrimeraClaseBase,
```

```
                                    public MiSegundaClaseBase
                                    { /* ... */ };
```

Otra posibilidad consistiría en dejar invariante la declaración de la clase que queremos promover a *persistente*, creando una nueva clase derivada de ésta y de *TStreamable*, como por ejemplo:

```
class MiClase [: [MiClaseBase [,OtrasClasesBase]*]]
{ /* ... */ };
class MiClasePersistente : public MiClase, public TStreamable
{ /* ... */ };
```

Naturalmente la penalización inherente a tal desarrollo consiste en la re-escritura de los constructores apropiados a la nueva clase. Tal enfoque aparece apropiado, sin embargo, para la aplicación de propiedades bien-definidas (como la que nos ocupa) en entornos *Smalltalk-like* en los que se enfatice el uso básico de los constructores por defecto (como por ejemplo los basados en la arquitectura MVC: *Model-View-Controller*), pues como quiera que, como es sabido, la no-intervención explícita en la inicialización de las clases base respecto de los constructores de una clase dada asegura el uso del constructor por defecto de aquéllas, tal aspecto redundaría en una deseable simplificación del código a la vez que encapsularía en una clase aparte los detalles específicos de persistencia dejando invariante el código ya escrito. Una tal reunión de clases persistentes por herencia múltiple resultaría en una *estructura paralela persistente* con respecto a las clases iniciales sobre la que, de nuevo, cabría aplicar la disyuntiva práctica establecida en el párrafo anterior.

Retomemos, salvado el inciso, la definición de la función *escritora*: nos ha faltado añadir el tratamiento de inserción en el objeto *opstream* de los miembros, incorporados o no, de tipo *agregado*, tales como *arrays, sets, listas encadenadas* y, en general, *contenedores* de objetos. Generalizando, tales casos habrían de reducirse a los ya vistos mediante la aplicación ordenada de la función *escritora* a cada uno de los objetos contenidos en el objeto de tipo *agregado*, accesibles a través de **iteradores** o, en un nivel más alto, de funciones diseñadas para recorrer apropiadamente los objetos. En los arrays de objetos (de tipos tanto incorporado como *definido-por-el-usuario*) el sistema no aporta *iterador* propiamente dicho, de forma que la navegación a través de los objetos contenidos en el array es responsabilidad del desarrollador. La subfunción *escritora* del array habrá de ser implementada, pues, como una estructura de control. Los objetos de tipo *agregado-definido-por-el-usuario* observarán, a los presentes efectos, el control de su propia función *escritora*. Veamos, pues, en la práctica, los distintos supuestos más detalladamente:

```
const MAXFILA = 2;
const MAXCOL = 4;
class PCliente;
class ArrayPersistente;
```

```
class MiClase {
        float floatArray[MAXFILA][MAXCOL]
        PCliente PClienteArray[MAXFILA][MAXCOL]
        ArrayPersistente miArrayPersistente;
        // ...
};
// ...
class MiClasePersistente :      public MiClase,
                                                virtual public TStreamable
                                                { /* ... */ };
void MiClasePersistente::write( opstream& ops )
{
        // ...
        for ( int m = 0; m<=MAXFILA; m++ )
                for ( int n = 0; n<= MAXCOL; n++ ) {
                        ops << floatArray[ m ][ n ];
                        ops << PClienteArray[ m ][ n ];
                }
        ops << miArrayPersistente;
        // ...
}
// ...
class ArrayPersistente: public Array, public virtual TStreamable
{ /* ... */ };
void ArrayPersistente::write( opstream& ops )
{
        ArrayIterator miIterador( *this );
        StringPersistente miStringPersistente;
        miIterador.restart();
        while ( miStringPersistente =
                        (StringPersistente*)miIterador++ )
                ops << miStringPersistente;
}
```

Hemos utilizado en la exposición las clases *Array* y *ArrayIterator*, incluidas en la Librería de Clases Contenedoras de Objetos (*Object Container Class Library*), un entorno *SmallTalk-like* incluido en la versión profesional de Borland C++ 3.1. Así, vemos que los detalles específicos de manipulación de los elementos internos de un objeto son tratados por métodos encapsulados en los mismos: esta es, en definitiva, la esencia del paradigma de OOP.

De acuerdo con lo expuesto, indicaremos que, básicamente, la operación de inserción de un objeto derivado de *TStreamable* en un objeto de tipo (o derivado de) *opstream* resulta en el siguiente código:

```
opstream& operator <<( opstream& ops, TStreamable& t )
{
        ops.writePrefix( t );           // [streamableName()
        ops.writeData( t );  // chequea registro de Clase
                                                // y escribe miembros internos
        ops.writeSuffix( t );           // ]
        return ops;
}
```

Pero, ¿qué ocurre cuando el objeto que pretendemos persistente contiene uno o más punteros a otros objetos, persistentes o no? Veámoslo detalladamente.

Imaginemos una clase conteniendo un dato miembro significado en un puntero a una clase no establecida como persistente. Como quiera que la persistencia que aquí tratamos es estática, la cualificación de un determinado objeto miembro como *transient* podría determinarse de forma indirecta por su no inclusión en la función *escritora*, en cuyo caso la única restricción será la de procurar la inicialización a cero de tales punteros en la construcción de los objetos contenedores de los mismos durante el proceso de *lectura*. Pero esto no resultaría muy conveniente, pues generaría una independencia entre los procesos de *lectura* y *escritura* difícil de documentar. Más apropiada sería la "nulificación" del puntero en la función *escritora*, o más explícitamente mediante la siguiente codificación:

```
// la siguiente línea aparecería en
//  cualquier lugar de <objstrm.h>
const int TRANSIENT = 0;
// …
void MiClasePersistente::write( opstream& ops )
{
        // …
        ops << ( punteroAObjetoNoPersistente = TRANSIENT );
}
```

La transcripción escritora de un puntero nulo sería, por otra parte, transparente, pero, ¿qué ocurriría cuando lo que se deseara insertar en un *opstream* fuera un puntero a un objeto persistente? Un tal objeto podría ser múltiplemente apuntado por otros objetos, de forma que si decidiéramos asociar cada aparición en un objeto de un puntero al mismo con una operación individual de inserción, el procedimiento devendría costosamente inefectivo. Una posible solución consistiría en la habilitación de un almacenamiento temporal donde fueran registrados los distintos objetos encontrados en la secuencia de inserción y antes de su escritura. A cada uno de tales objetos, representados por sus direcciones en memoria, se les asociaría un *índice* o contador, significativo de su orden de registro en tal almacenamiento, de forma que, dado un puntero a un objeto y antes de proceder a su escritura, se buscaría en el objeto contenedor o almacenador el par (*direccion*, *indice*) que casara con la dirección contenida en el puntero dado, resultando en la siguiente bifurcación: si el objeto apuntado no estuviera registrado (esto es, si no existiera el par con clave *direccionDeBusqueda*), se procedería a su registro e inserción, desreferenciando el puntero, en el *pstream*; si, por el contrario, se encontrara registrado se procedería a escribir la referencia ordinal del mismo indicando su situación como entidad en la escala secuencial escritora. Examinemos seguidamente la implementación que Borland realiza de este esquema.

En la librería que nos ocupa el objeto almacenador es un contenedor de tipo *TPWrittenObjects*, una clase derivada públicamente de *TNSSortedCollection*, que a su vez representa un contenedor ordenado de objetos de clases no derivadas de *TStreamable* (no olvidemos que lo que se almacenará será la representación no-persistente de los objetos persistentes a insertar en el *pstream*). En el objeto de tipo *TpWrittenObjects* se insertarán objetos *TPWObj*, encapsuladores de los distintos pares (*direccion*, *indice*) construidos a partir de los objetos secuencialmente dispuestos para ser escritos en el *pstream*. Este mecanismo incorpora, además, con relación al expuesto en el párrafo anterior, un nivel adicional de optimización, pues los objetos persistentes contenidos en objetos a insertar en *opstream*s son también registrados y referencialmente insertados como objetos *TPWObj* en la base de datos: de esta forma todos los objetos persistentes ya *escritos* y apuntados por punteros no serán duplicados en el *pstream*. Esto sugiere que el código para el registro de bs objetos habrá sido desplazado desde el bloque de inserción de punteros hasta el bloque de inserción de objetos y, de hecho, se encuentra en el cuerpo de la función *void opstream::writeData(TStreamable&)*.

La casuística de inserción de punteros a objetos persistentes es tratada de la siguiente forma: en la sección *public* del cuerpo de descripción de la clase *pstream* se declara:

```
enum PointerTypes { ptNull, ptIndexed, ptObject };
```

siendo utilizados tales enumeradores como señalizaciones en el *pstream* de la modalidad de escritura utilizada. Examinemos el código:

```
opstream& operator << ( opstream& ops, TStreamable *t )
{
        unsigned index;
        if( t == 0 )                    // puntero nulo ó puntero
                                        //  a objeto Transient
                ops.writeByte( pstream::ptNull );
        // seguidamente se busca un TPWObj conteniendo
        //  (void*) address == t
        //  Si la búsqueda tiene éxito se devuelve el
        //  ordinal de escritura del objeto
        else if( (index = ps.find( t )) != 0 ) {
                ops.writeByte( pstream::ptIndexed );
                // se escribe el índice orginal
                ops.writeWord( index );
        } else {    // inserción normal como objeto persistente
                ops.writeByte( pstream::ptObject );
                ops << *t;
        }
        return ops;
}
```

Esta estructuración permitirá la adecuada extracción en su momento de los objetos alojados en el *pstream*.

Nos resta únicamente considerar un aspecto: debe establecerse un mecanismo de seguridad que compruebe si el sistema posee el código adecuado para la recuperación o extración de cada uno de los objetos insertados en el *pstream*. Como veremos al considerar la función *lectora*, en las clase persistentes debe añadirse el código general

```
TStreamableClass RegPClase (        "TPClase",
                                        TPClase::build,
                                        __DELTA( TPClase ) );
```

el cual originará el registro de la clase (indexado por medio de *streamableName()*) en un objeto de clase *TStreamableTypes*, que crearía si fuera necesario, y que sería asignado al puntero *static* identificado por *types* en *pstream*. De esta forma se generaría una base de datos conteniendo todas las clases deseadas persistentes de una aplicación.

FUNCION LECTORA

De igual forma a como hemos procedido con la función *escritora* debemos implementar la función *lectora*, declarada en la sección *public* de nuestra clase:

```
void* MiClasePersistente::read( ipstream& ips ) { /* … */ }
```

donde *ips* es una referencia a un objeto de clase *opstream (input persistent stream)*, derivada de *pstream* para operaciones de extracción. La única restricción en la definición de tal función es la obligatoriedad de conservar el orden de aparición de los distintos elementos a ser leídos tal como fue implementado en el cuerpo de la función *escritora*. Ello se debe, evidentemente, al hecho que tales elementos son almacenados de forma secuencial. Realmente la función *lectora* es una cuasi-repetición formal de la función *escritora*, sustituyendo *opstream&* por *ipstream&*, *write* por *read*, y trocando el tipo de retorno *void* por *void** (y añadiendo a tal efecto la sentencia de retorno *return this;*).

Atisbamos, de esta forma, un primer procedimiento de mecanización generativa de código. En lugar de escribir repetitivamente dos funciones completas, podríamos definir en la clase una función de propósito general que después aplicaríamos convenientemente en sus apropiadas funciones *escritora* o *lectora*. Veámoslo:

```
class TStreamable {
protected:
        void write( opstream& ops ) { persist( ops ); }
        void* read( ipstream& ips ) { persist(ips);return this; }
        virtual void persist( pstream& ps ) = 0;
        // …
```

```
};

class MiNuevaClasePersistente:
        public MiClaseBase, public TStreamable { /* ... */ };
void MiNuevaClasePersistente::persist( pstream& ps )
{
        MiClaseBase::persist( ps );
        ps[ objetoDeTipoIncorporado ];
        ps[ objetoDeOtraClasePersistente ];
        ps[ punteroAObjetoDeOtraClasePersistente ];
}
```

Notemos que se ha modificado la clase *TStreamable*, incorporando una nueva función virtual pura (*persist*) de obligatoria redefinición en sus clases derivadas, más la definición basada en ésta de las funciones *read* y *write*. Se han trocado, también, estas últimas funciones desde virtuales puras a métodos normales de la clase, siendo así normalmente heredadas, sin necesidad de redefinición, por las clases persistentes derivadas de *TStreamable*. Necesitaríamos, por otra parte, definir sobrecargas del operador '**[]**' en las clases *opstream* e *ipstream*, representadas genéricamente de la siguiente forma:

```
class ipstream : virtual public pstream {
public:
        pstream& operator [] ( tipo& objetoDeTipo)
        {
                return *this >> objetoDeTipo;
        }
        // ...
};

class opstream : virtual public pstream {
public:
        pstream& operator [] ( tipo& objetoDeTipo )
        {
                return *this << objetoDeTipo;
        }
        // ...
};
```

Se procurará, así, una sobrecarga de tal operador para cada una de las sobrecargas existentes en tales clases para los operadores de inserción y extracción. Estamos obviando, empero, (y de aquí el *cuasi* en la pretendida repetición formal de código) la especial codificación necesitada para el archivo y recuperación de objetos de tipo *agregado-definido-por-el-usuario*. Más adelante concretaremos en detalle esta idea.

En el acercamiento práctico a la persistencia influyen sobre todo las consideraciones sobre la recuperación de los objetos, constituyéndose las restricciones observadas en tal proceso en límites del modelado de su almacenamiento, que dependerá de aquél. Hemos repasado comprehen-

sivamente el proceso de escritura de los datos internos de los objetos en *streams persistentes*, pero ¿qué ocurre con la identidad de los objetos tal y como fue establecida en el principio del anexo? Antes de iniciar el volcado de los datos miembro se significa en el *pstream* el identificador de la clase (devuelto por la función *StreamableName()*) antecedido por un carácter '[' (tras el que, realmente, se escribe el valor de la longitud de la cadena, moldeado como *char*). Tal señalización nos habrá de servir para crear en memoria los objetos de la clases especificadas, para después proceder a la copia recursiva desde el *pstream* de los datos miembro. Esto requiere un detenido examen: veamos la representación simbólica de un *pstream* tras la aplicación, según lo expuesto, de la función *escritora* a un objeto de clase *CID1*:

```
[CID1itdv11|itdv12[CID2itdv21]itdv13[CID3]
        [CID4itdv41itdv42itdv43]itdv14]
```

donde ***CIDn*** representa un identificador de clase devuelto por la función *StreamableName()*, mientras que ***itdvnx*** equivale al valor de los datos de tipo incorporado contenidos en cada uno de los objetos de tipo *CIDn*, mientras que el par **[]** deviene en alfabeto de un lenguaje restringido de Dyck. La reconstrucción de la estructura de objetos transcrita al *pstream* aparece engañosamente transparente, como puede apreciarse en el siguiente código simbólico:

```
CD1* punteroCD1 = new CD1;
punteroCD1->idt11 = idtv11;
punteroCD1->idt12 = idtv12;
        CD2* punteroCD2 = new CD2;
        punteroCD2->idt21 = idt22;
punteroCD1->CD2td = *punteroCD2;
punteroCD1->idt13 = idtv13;
        CD3* punteroCD3 = new CD3;
punteroCD1->CD3td = *punteroCD3;
        CD4* punteroCD4 = new CD4;
        punteroCD4->itd41 = itdv41;
        punteroCD4->itd42 = itdv42;
        punteroCD4->itd43 = itdv43;
punteroCD1->CD4td = *punteroCD4;
punteroCD1->itd14 = itdv14;
```

¿Realmente es errónea esta sencilla codificación? Bien: debemos notar, ante todo, que la correspondencia entre código y *pstream* es perfecta: tal código nos sirve <u>únicamente</u> para su aplicación a ese único *pstream*, por lo que parece que deberíamos asociar un bien determinado método a cada *pstream*. ¡Pero esto es absurdo!: lo que deseamos implementar es la capacidad general de recuperación a memoria de los objetos almacenados en un *pstream* con independencia de la identidad concreta de estos objetos. Examinemos, por otro lado, el "inocente" código:

```
CD3* punteroCD3 = new CD3;
```

En él estamos utilizando el operador *new* para alojar en el área de memoria de almacenamiento libre un nuevo objeto del tipo *CD3*, inicializado utilizando el constructor por defecto de la clase *CD3*. Imaginemos que en tal constructor aparece el siguiente código:

```
CD3::CD3() : MiWindowEstandar()
{
        // inicialización de variables diversas
        iniciaDialogoModal();
}
```

Así la invocación del constructor resultaría en la interrupción de toda la secuencia de *lectura*, al crear el objeto una ventana estándar e iniciar un diálogo modal.

Analizando las trabas expuestas observamos que necesitamos, pues, un código general, que habrá de ser usado por lo que Borland denomina ***stream manager***, y que nos permitirá extraer de un *pstream* dado, cuyo contenido es desconocido en tiempo de compilación, información suficiente para reconstruir un sistema interrelacionado de objetos con determinados estados internos. De hecho, la definición en cada clase de la función *lectora* proporciona el más adecuado método de restauración de los datos internos de los objetos de tal clase, pero piénsese que C++ es un *lenguaje de jerarquía dual*: las clases son entidades distintas de los objetos, de forma que el mensaje de lectura tiene que ser dirigido al objeto, el cual no existe antes de la aplicación a la clase de un determinado constructor cuya idoneidad desconocemos. Aclaremos la situación: en primer lugar y antes de extraer del *pstream* los valores de los datos miembros de un determinado objeto debemos proceder a la creación de éste, pero no tenemos forma de saber qué constructor debemos utilizar a fin que no interfiera con la secuencia de lectura o genere indeseables efectos laterales, como por ejemplo el envío de mensajes que alterarían la estructura interna de objetos ya reconstruidos. Necesitamos, pues, de un constructor especial para cada clase que se limite a alocar el espacio libre necesario para alojar el esqueleto y la *tabla de funciones virtuales* de un nuevo objeto de tal clase que, como contenedor, será oportunamente "rellenado" desde el *pstream* (recordemos aquí que el tamaño de un objeto no es igual a la suma del tamaño de sus miembros: los objetos de una clase vacía, por ejemplo, poseen un tamaño no-nulo). Tal constructor debe poseer características identificadoras únicas, un cuerpo vacío y un sistema de control de la inicialización de sus clases base. Todo esto se cumple, en definitiva, con el constructor

```
class MiClasePersistente :      public MiClaseBasePersistente,
                                                virtual public TStreamable {
public:
        MiClasePersistente( StreamableInit s );
        // ...
};
MiClasePersistente::MiClasePersistente( StreamableInit s )
```

```
    : MiClaseBasePersistente( streamableInit ) {}
```

donde *StreamableInit* es un *enum* que consta del único enumerador *streamableInit*, declarado en <objstrm.h> (o en <ttypes.h> para *TurboVision*) de la forma

```
enum StreamableInit { streamableInit };
```

Vemos que la singularidad del enumerador asegura una cierta protección contra el mal uso del constructor, a la vez que se explicita el uso de similar constructor en las clases base a las que se habría aplicado el mismo tratamiento de persistencia. Nótese que no es necesaria la inicialización expresa de *TStreamable*, pues es ésta la única clase que el *stream manager* directamente conoce, asegurando que la aplicación rutinaria del constructor por defecto no vulnerará el proceso. Hay que destacar, no obstante, que si la clase *aspirante-a-persistente* derivara, entre otras, de una o más clases en las que no hubiera sido implementado este tipo de persistencia, no sería posible acceder a estos constructores especiales, y el orden de aplicación de constructores en derivación podría generar el mismo problema que se intentaba evitar: la intervención incontrolada de un constructor (normalmente por defecto) de una clase base en el proceso de creación del esqueleto del objeto. Notamos, entonces, que el mantenimiento de la seguridad del proceso nos obligaría a repetir o desplazar la derivación de persistencia hacia atrás en la escala jerárquica de la clase inicialmente elegida. En realidad tal circunstancia refuerza la idoneidad, dada una jerarquía de clases, de la aplicación de la derivación de *TStreamable* cuando menos (y tal vez únicamente) a sus clases base.

Poseemos ya, según lo visto, un adecuado esquema de constructores, pero ¿quién efectuará las correspondientes llamadas a los mismos? La respuesta resulta en otra pregunta: ¿quién conoce más de un objeto que el objeto en sí? Efectivamente: la tarea de creación de esqueletos de objetos debería ser encomendada a uno de tales objetos. Debemos, pues, declarar en la sección *public* de nuestra clase una función constructora que podría ser definida así:

```
TStreamable* MiClasePersistente::build()
{
        return new MiClasePersistente( streamableInit );
}
```

Pero, ¿puede un objeto crear otro objeto u operar de alguna manera antes de ser él mismo creado?. Evidentemente no. Necesitamos aquí, así, una operación de creación no encapsulada en un objeto particular, pero restringida al ámbito de la descripción de tipo del mismo: esto es, una función miembro estática. La función *build()* sería declarada, pues, como *static* en el protocolo de descripción de la clase afectada, siendo accedida a través de un puntero, accesible a su vez en la base de datos soporte del registro de clases por medio del identificador devuelto por *streamableName()*. Recapitulemos: como ya vimos cuando detallábamos la

función *escritora*, es preciso añadir para cada una de las clases deseadas persistentes una línea de código, creadora de un objeto de tipo *TStreamableClass*, la cual registrará convenientemente la clase en un contenedor especial. Veámoslo aplicado a nuestra clase ejemplo:

```
TStreamableClass RMiClasePersistente(
                    MiClasePersistente::name,
                    MiClasePersistente::build,
                    __DELTA( MiClasePersistente ) );
```

El prototipo del constructor de *TStreamableClass* es el siguiente:

```
public:TStreamableClass::TStreamableClass(
                    const char* identificadorDeClase,
                    TStreamable* ( _FAR *punteroAFuncionBuild )(),
                    int offsetTStreamable );
```

correspondiendo el tercer argumento al offset desde la base del objeto hasta el inicio de la parte *TStreamable* del mismo adquirida por derivación, cuyo valor será almacenado en un dato miembro de *TStreamableClass* llamado *delta*. Está distancia será convenientemente minorada de la dirección del objeto implícitamente moldeado a *TStreamable* en su uso iterativo como argumento de funciones llamadas desde su inserción o extracción en un objeto *pstream*. Este mecanismo permitirá el registro de los objetos de la clase encontrados en la secuencia *lectora* con su dirección correcta, y no con la correspondiente a su parte *TStreamable* (recordemos que los objetos son registrados como pares (*direccion*, *indiceSecuencial*)). En lugar de tener que calcular tal valor para cada clase, Borland provee una macro, ***__DELTA-(IdentificadorClase)***, que automáticamente realiza este trabajo. En definitiva, el código anteriormente descrito crea un nuevo objeto de tipo *TStreamableClass*, encapsulador de los valores asumidos por los argumentos de su constructor, el cual a su vez registrará la clase en un objeto contenedor de tipo *TStreamableTypes* apuntado por *pstream::types* (si el puntero es nulo, se creará un nuevo objeto contenedor y se asignara su dirección a aquél). En tal contenedor se registrarán todas las clases de una aplicación mediante la inserción de los pares (*identificadorClase*, *punteroAFuncion-BuildDeLaClase*) correspondientes a cada una de ellas.

Precisamente este mecanismo de registro obligará a explicitar el enlace con clases cuya declaración aparezca en módulos distintos del dado. A tales efectos Borland provee, igualmente, una macro cuyo cometido es proporcionar una referencia al objeto de tipo *TStreamableClass* usado para el registro de todas las clases de la aplicación, con la sintaxis siguiente:

```
__link(RegIdentificadorDeMiClasePersistenteExternaAEsteModulo)
```

Retomemos, al fin, la implementación en nuestra clase de la capacidad de recuperación de objetos a memoria desde un *pstream*. Como ya notamos, debe dotarse a cada clase de una función *lectora* de la forma:

```
void* MiClasePersistente::read( ipstream& ips )
{
        MiClaseBase::read( ips );
        ips >> objetosDeTipoIncorporado;
        ips >> objetoDeOtraClasePersistente;
        ips >> punteroAObjetoDeOtraClasePersistente;
}
```

estructural y secuencialmente paralela a la función *escritora*. Examinemos seguidamente el proceso de extracción de objetos significando mediante pseudo-código la secuencia lectora del *pstream*:

```
extracciónObjeto: lectura de la señal '[' de inicio de un objeto
lectura de la longidud (int) del identificador de clase del objeto
lectura del identificador de la clase MCP del objeto
búsqueda del identificador de clase en el registro
        de clases persistentes (precondición)
construcción de un nuevo objeto del tipo MCP usando el puntero a MCP::build
registro nuevo objeto MCP en objeto tipo TPReadObjects
llamada a la función apuntada por MCP::read
loop while  existen líneas de código en función MCP::read
        [ inserción de un objeto de tipo incorporado ]*
                lectura directa del valor de la variable desde el pstream
        [ inserción de un objeto de tipo definido-por-el-usuario ]*
                call extracciónObjeto
        [ inserción de un puntero a un objeto de tipo definido-por-el-usuario ]*
                select  señalizador puntero
                        case pstream::Null:
                                asignación de cero al puntero
                        case pstream::ptIndexed:
                                lectura índice registro orden lectura objeto
                                búsqueda por índice en registro objetos leídos
                                asignación dirección objeto encontrado a puntero
                        case pstream::ptObject:
                                call extracciónObjeto
                                asignación dirección nuevo objeto a puntero
                endselect
endloop
terminaInserciónObjeto: lectura del señalizador final del objeto ']'
```

Así como la función *escritora* usaba de un contenedor *TPWrittenObjects* para el control de los objetos ya escritos, la función *lectora* usa de un parecido objeto de tipo *TPReadObjects*, una clase derivada públicamente de *TNS-Collection*, que a su vez representa un contenedor no-ordenado de objetos de clases no derivadas de *TStreamable*.. En el objeto de tipo *TPReadObjects* se insertarán las direcciones en memoria de los objetos ya leídos, asimilando cada una al índice (base 1) secuencial de registro, equivalente al de *escritura*. En correspondencia con el mecanismo *escritor*, el código para el registro de los objetos habrá sido insertado en el cuerpo de la función *void ipstream::readData(TStreamable&)*. Efectivamente el seguimiento de punteros se ha traspasado al almacenamiento secundario en una separación dual que,

lógicamente, originará una penalización sobre el más eficiente mecanismo transparente de *nivel singular*.

Como vemos, el modelado de la función *escritora* sobre la base de los procedimientos *lectores* origina una recíproca correspondencia autoexplicativa del proceso de extracción con respecto al de inserción. Como vimos anteriormente, esto nos induce a usar mecanismos más generales, como el expuesto mediante sobrecarga del operador '**[]**'. En aquel momento dejamos pendiente la revisión de la generalización en la persistencia de ciertos objetos de tipo *agregado*. Veámoslo ahora.

La aplicación más inmediata de lo expuesto tendría lugar en la jerarquía de clases contenedoras notadas por Borland como "collections". Estas clases llevan incoporado código soporte para la persistencia expuesta, habiendo solucionado el problema del almacenamiento y recuperación de objetos con un mecanismo de tipo-seguro que consiste en la implementación en cada clase de los métodos *writeItem()* y *readItem()*, inicialmente definidos como funciones virtuales puras en *TCollection* de la forma:

```
virtual void writeItem( void*, opstream& ) = 0;
virtual void* readItem( ipstream& ) = 0;
```

forzando su redefinición en las clases de ella derivadas. Simplificando, este mecanismo exigiría, en una clase ejemplo *Contenedora*, la explicitación de tipo de los items de la siguiente manera:

```
void Contenedora::writeItem( void *prst, opstream& ops )
{
        ops << (Contenedora*)prst;
}

void* Contenedora::readItem( ipstream& ips )
{
        Contenedora *prst;
        is >> prst;
        return prst;
}
```

Una distinta aproximación metodológica a la construcción lectora expuesta sería, en C++, la basada en el "Idioma de Ejemplares" detallado en [Copli92], inmersa en las más generales técnicas de "constructores virtuales". Tales técnicas merecen, empero, capítulo expositivo aparte.

EJEMPLOS PRÁCTICOS

Borland ofrece una pedagógica ejemplificación de la aplicación de la persistencia en una jerarquía de objetos "gráficos" en el archivo de demostración *tvguid021.cpp*, incluido en *TurboVision*. A pesar que el código es extremadamente simple, el ejemplo no podría calificarse de muy apropiado pues, tras su compilación y enlace, el observador poco avezado, atónito ante la repetición de una composición gráfica estática, podría preguntarse ¿para qué demonios sirve esto? Una aproximación más interesante hubiera sido la implementación de persistencia en, por ejemplo, un objeto *port* adecuado a la *client area* de una determinada ventana, sobre el que se utilizarían objetos *lápices*, *brochas* y *rodillos*, junto con *colores*, *fuentes tipográficas*, etc. mediante una transformación, gestionada por un objeto *transformador xvirtual*, desde coordenadas virtuales. Estaríamos operando, de hecho, con colores y formas en un mapa de bits. La simplificada persistencia de éste causaría que, en cualquier momento, pudiéramos "archivarlo" junto con las relaciones complejas que contuviera (incluyendo el último "dibujo" realizado) y fácilmente después recuperarlo, de tal forma que, en estructuras complejas inter-relacionadas como ésta, la secuencia concreta de transferencia entre el archivo y los objetos es desconocida por el usuario o desarrollador.

La persistencia de objetos permite, entre otras aplicaciones, soportar el almacenamiento expedito de imágenes, sonidos y, en general, composiciones multimedia en ODBMSs.

PERSISTENCIA RECURSIVO-DERIVATIVA (RDP)

Como hemos visto, la implementación en una clase dada del tipo de persistencia expuesto obedece a la incorporación en la descripción de tal clase de determinados datos y funciones miembros. En realidad, como ya apuntamos, tal mecanismo es susceptible de ser mecanizado. Pero enfoquemos el problema desde una óptica más general: necesitaríamos de un esquema que permitiera la cualificación o no de un objeto como persistente en tiempo de ejecución. Esto nos sugiere que debiera ser añadido un filtro interruptor a cada una de las clases en una aplicación, de forma que soportaran métodos para responder a este tipo de mensajes. Lo más evidente sería implementar la cualidad general de persistencia en todas las clases y añadir a éstas un miembro o flag que filtrara la acción del *stream manager*.

Deben ser añadidas, antes de nada, las siguientes modificaciones a las clases que se notan, señaladas en negrita:

```
class TStreamable {
        friend class pstream;
private:
        persist_mode persistStatus;
```

```
protected:
        TStreamable() {
                persistMode = PERSISTENT;
        }
        enum persist_mode { TRANSIENT, PERSISTENT };
        virtual void persist( pstream& ) = 0;
        void* read( ipstream& ips ) { persist(ips);return this; }
        void write( opstream& ops) { persist( ops ); }
public:
        boolean isTransient() { return persistStatus == TRANSIENT; }
        void setPersistMode( persist_mode pmode ) {
                persistStatus = pmode; }
        // …
};

class  pstream {
public:
        virtual pstream& operator [] ( signed char& ) = 0;
        virtual pstream& operator [] ( unsigned char& ) = 0;
        virtual pstream& operator [] ( signed short& ) = 0;
        virtual pstream& operator [] ( unsigned short& ) = 0;
        virtual pstream& operator [] ( signed int& ) = 0;
        virtual pstream& operator [] ( unsigned int& ) = 0;
        virtual pstream& operator [] ( signed long& ) = 0;
        virtual pstream& operator [] ( unsigned long& ) = 0;
        virtual pstream& operator [] ( float& ) = 0;
        virtual pstream& operator [] ( double& ) = 0;
        virtual pstream& operator [] ( long double& ) = 0;
        virtual pstream& operator [] ( TStreamable& ) = 0;
        virtual pstream& operator [] ( void* ) = 0;
        // …
};

class ipstream : virtual public pstream {
public:
        pstream& operator [] ( signed char& sch ) {
                return *this >> sch;
        }
        pstream& operator [] ( unsigned char& uch ) {
                return *this >> uch;
        }
        pstream& operator [] ( signed short& ssh ) {
                return *this >> ssh;
        }
        pstream& operator [] ( unsigned short& ush ) {
                return *this >> ush;
        }
        pstream& operator [] ( signed int& sint ) {
                return *this >> sint;
        }
        pstream& operator [] ( unsigned int& uint ) {
                return *this >> uint;
        }
        pstream& operator [] ( signed long& slng ) {
```

```
                return *this >> slng;
        }
        pstream& operator [] ( unsigned long& ulng ) {
                return *this >> ulng;
        }
        pstream& operator [] ( float& fl ) {
                return *this >> fl;
        }
        pstream& operator [] ( double& dbl ) {
                return *this >> dbl;
        }
        pstream& operator [] ( long double& ldbl ) {
                return *this >> ldbl;
        }
        pstream& operator [] ( TStreamable& prst ) {
                return *this >> prst;
        }
        pstream& operator [] ( void* prst) {
                return *this >> prst;
        }
        // …
};

class opstream : virtual public pstream {
public:
        pstream& operator [] ( signed char& sch ) {
                return *this << sch;
        }
        pstream& operator [] ( unsigned char& uch ) {
                return *this << uch;
        }
        pstream& operator [] ( signed short& ssh ) {
                return *this << ssh;
        }
        pstream& operator [] ( unsigned short& ush ) {
                return *this << ush;
        }
        pstream& operator [] ( signed int& sint ) {
                return *this << sint;
        }
        pstream& operator [] ( unsigned int& uint ) {
                return *this << uint;
        }
        pstream& operator [] ( signed long& slng ) {
                return *this << slng;
        }
        pstream& operator [] ( unsigned long& ulng ) {
                return *this << ulng;
        }
        pstream& operator [] ( float& fl ) {
                return *this << fl;
        }
        pstream& operator [] ( double& dbl ) {
                return *this << dbl;
```

```
        }
        pstream& operator [] ( long double& ldbl ) {
                return *this << ldbl;
        }
        pstream& operator [] ( TStreamable& prst ) {
                return *this << prst;
        }
        pstream& operator [] ( TStreamable* prst) {
                if ( prst->isTransient() )
                        prst = TRANSIENT;
                return *this << prst;
        }
        // ...
};
```

Nuestra "propuesta" se basará directamente en el esquema desarrollado por Borland, con las ventajas e inconvenientes que esto supone: ventajas como la disponibilidad inmediata del soporte estructural o el bien-testeado mecanismo funcional; inconvenientes como la penalización en tiempo de ejecución resultante del nivel adicional de indirección, o la menor flexibilidad resultante de la dependencia de un código dado.

Necesitaremos de un Metasistema que nos permita acoplar la recursividad de acceso sobre cada uno de los miembros de una clase, a la vez que "instale" y "mantenga" las características de persistencia de las clases.

Entre los diferentes sistemas a elegir, quizás el más evidente sea el que usaría de un preprocesador, aplicable como filtro sobre todas y cada una de las clases intervinientes en la aplicación deseada.

El preprocesador actuaría sobre el código original que a continuación se detalla:

```
class NombreClase [: [[cualificador]claseBase]+] {
[cualificador:]*
        // datos miembros
        tipoIncorporado tipoBasico;
        NombreClase1 objeto;
        NombreClase2* punteroAObjeto;
        tipoIncorporado arrayDeTiposBasicos[ max1 ];
        NombreClase2 arrayDeObjetos[ max2 ][ max3 ]
        // funciones miembro
        [valorRetorno funcionMiembro(argumentos);]*
};
```

produciendo la siguientes modificaciones, señaladas en negrita:

```
#ifdef PERSIST

#ifndef PERSIST_FLAG
#define PERSIST_FLAG
#       define Uses_pstream
```

```
#       define Uses_ofpstream
#       define Uses_ifpstream
#       define Uses_TStreamableClass
#       include <tv.h>
#endif  /* PERSIST_FLAG */

__link( RegClaseBase )
__link( RegNombreClase1 )
__link( RegNombreClase2 )
__link( RegNombreClase3 )

class NombreClase :         public TStreamable
                                    [,[[cualificador]ClaseBase]+] {
#else   /* PERSIST */
class NombreClase [: [[cualificador]claseBase]+] {
#endif  /* PERSIST */
[cualificador:]*
        // datos  miembros invariantes
        // funciones miembro invariantes
#ifdef PERSIST
private:
        int indiceImpulsion;
        virtual const char* streamableName() const {
                return name; }
protected:
        NombreClase( StreamableInit ) :
                ClaseBase( streamableInit ) {}
        void persist( pstream& );
public:
        static const char* const name = "NombreClase";
        static TStreamable* build() {
                return new NombreClase( streamableInit ); }
#endif  /* PERSIST */
};

#ifdef PERSIST
inline ipstream& operator >> ( ipstream& ips, NombreClase& nc )
{       return ips >> (TStreamable&)nc; }
inline ipstream& operator >> ( ipstream& ips, NombreClase*& nc )
{       return ips >> (void *&)nc; }
inline opstream& operator << ( opstream& ops, NombreClase& nc )
{       return ops << (TStreamable&)nc; }
inline opstream& operator << ( opstream& ops,NombreClase*& nc )
{       return ops << (void *&)nc; }

void NombreClase::persist( pstream& ps )
{
        int indice[ 2 ];    // nº máximo de dimensiones en arrays
        ClaseBase::persist( ps );
        ps[ tipoBasico ];
        ps[ objeto ];
        ps[ punteroAObjeto ];
        ps[ referenciaAObjeto ];
        for ( indice = 0; indice <= max1; indice++ )
```

```
                    ps[ arrayDeTiposBasicos[ indice[0] ] ];
            for ( indice[0] = 0; indice[0] <= max2; indice[0]++ )
                    for (indice[1] = 0; indice[1] <= max3; indice[1]++)
                            ps[arrayDeObjetos[ indice[0] ][ indice[1] ]];
}

TStreamableClass RNombreClase ( RNombreClase::name,
                                RNombreClase::build,
                                __DELTA( RNombreClase ) );

#endif  /* PERSIST */
```

Tal preprocesador -al que en adelante denominaremos **CP5** (*C Plus Plus Persistence Pre-Processor*)- deberá registrar, adicionalmente, la descripción de cada clase en una database aparte. ¿Para qué? Bien: imaginemos que la clase de un determinado objeto sufre una variación tras el archivo en un *pstream* del mismo. CP5 habrá de ser instruido para comparar cada clase con la descripción registrada de la misma, y si aprecia algún cambio con respecto a ésta, buscará en el código una función constructo-conversora del tipo:

```
NuevaClaseModificada(       AntiguaDescripcionClase
                                objetoAntiguaClase );
```

la cual, básicamente, explicitaría la transferencia de identidades desde una descripción a otra (la no existencia expresa de tal método obligaría a CP5 a suplir un conversor por defecto, no siempre apropiado). Tal conversor podría ser utilizado bien para la actualización de los *pstream*s (mediante la compilación temporal de ambas descripciones) bien para su inclusión en un sistema de *versioning* de clases, similar al usado en ODBMSs.

La construcción en C++ del preprocesador CP5, usando quizá de la técnica llamada de "*constuctores virtuales*" notada en [Copli92] y [Eckel92], excede el objetivo marcado en la redacción de este anexo, cual inicialmente fue la exposición, a un nivel intermedio, de determinadas técnicas no complejas de implementación de la persistencia en sistemas de objetos. La codificación de CP5 pudiera, empero, merecer por su interés (piénsese en analizadores léxicos, scanners, etc.) un próximo trabajo detallado.

LA EVOLUCIÓN DE LA PERSISTENCIA

Bien, lo que aquí mayormente se ha detallado es el esquema de persistencia de un compilador concreto: Borland C++ 3.1. Pero, ¿se trata de una estructuración inamovible? Ciertamente no. En la versión 4.0 de este compilador, una decisión de diseño, cual es el cambio de las librerías de clases de contenedores desde una jerarquía cósmica (tipo Smalltalk) a una parametrización completa mediante plantillas, repercute grandemente sobre la implementación de la persistencia, aunque conserva en buena medida su interfaz. Así, por ejemplo, desaparecen las sobrecargas de los operadores:

```
opstream& operator << ( opstream&, TStreamable* );
ipstream& operator >> ( ipstream&, void* & );
```

a la vez que se solventa la cualificación de clases persistentes mediante el uso de las macros DECLARE_STREAMABLE e IMPLEMENT_STREAMABLEX, donde X es un número que depende de las clases bases inmediatas y de las clases bases virtuales de que deriva nuestra clase candidata. De alguna manera este cambio en la implementación afecta de manera menor a nuestro código. Bien: esto es OOP. Hay que señalar, por último, que el esquema detallado en el presente anexo es, con las lógicas variaciones, el más utilizado en las librerías comerciales de clases.

REFERENCIAS

[Atwoo91] Thomas Atwood, *At last! A distributed database for Windows 3.0*, Object Magazine, 1(1), 1991.

[Booch92] Grady Booch & Michael Vilot, *Physical design: persistence*, C++ Report, 4(3), 1992.

[Borla91] Borland International Inc, *Borland C++ 3.1 Programmer's Guide, TurboVision for C++, ObjectWindows for C++*, 1992.

[Camm91] Stephanie J. Cammarata & Christopher Burdof, *PSE: and object-oriented simulation environment supporting persistence*, Journal of Object-Oriented Programming, 4(6), 1991.

[Devis94] Ricardo Devis Botella, *OOA/OOD : Un Enfoque Práctico*, En preparación.

[Copli92] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1992.

[Eckel92] Bruce Eckel, *Virtual constructors*, C++ Report, 4(3), 1992.

[Gorle90] Keith Gorlen, S. Orlow & P. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley, New York, 1990.

[Meyer88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.

[Rumba91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy & William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[Urloc91] Zack Urlocker, *From applications to frameworks*, Hotline on Object-Oriented Technology, 2(11), 1991.