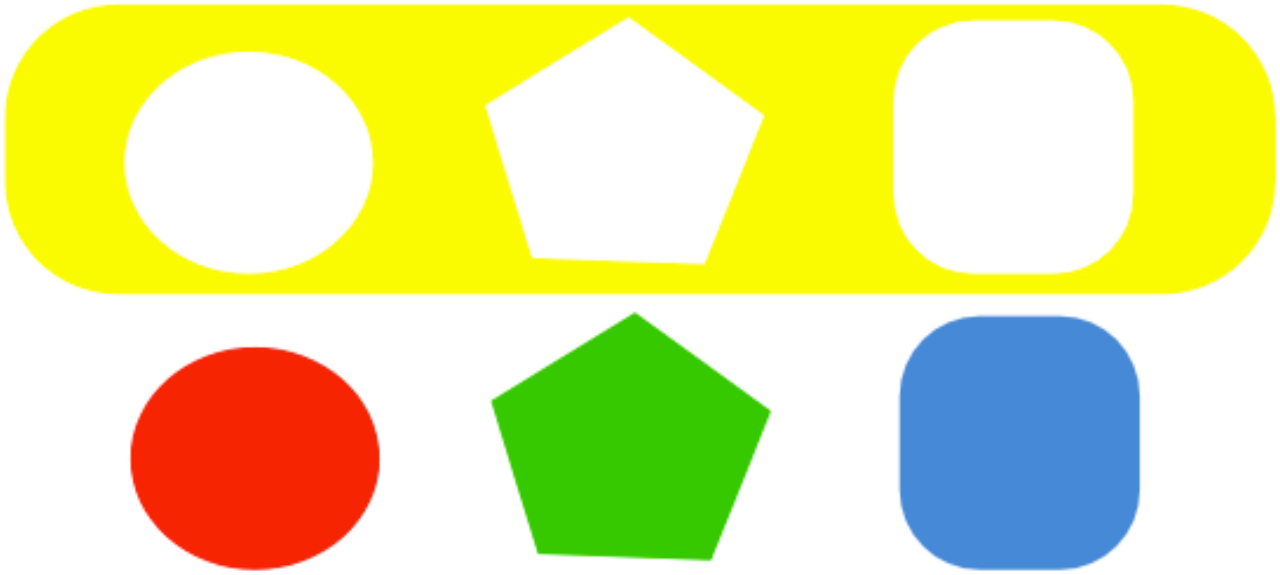
Programación Orientada a Objetos



Roberto Rodríguez Echeverría
Encarna Sosa Sánchez
Álvaro Prieto Ramos

PROGRAMACIÓN ORIENTADA A OBJETOS

Programación Orientada a Objetos por Roberto Rodríguez Echeverría, Álvaro Prieto Ramos, Encarna Sosa Sánchez se encuentra bajo una Licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 3.0 Unported.
Basada en una obra en www.libreriaalvaro.com/libropoo.pdf.

ISBN: 84-609-0003-7
Depósito Legal: CC-11-2004

Prólogo

Desde principio de la década de los noventa hasta la actualidad, la Programación Orientada a Objetos se ha establecido como el paradigma más extendido entre la comunidad de programadores. Las asignaturas de programación de las Facultades y Escuelas de Informática no han sido ajenas al uso de este paradigma de programación. Estas asignaturas, normalmente, se han basado en buenos y completos libros de expertos en la materia. No en vano, existe una amplia literatura acerca del tema. Pero quizás, esta amplia literatura provoca, que, tanto profesores como alumnos, o incluso, personas externas al ámbito académico interesadas en este paradigma, se vean desbordados ante la cantidad de material disponible, del cual, además, posiblemente necesiten sólo una pequeña parte.

La intención de los autores al escribir este libro es la de cubrir una laguna que, en nuestra opinión, existe en la vasta literatura acerca del tema: la falta de una guía accesible y concisa de introducción a los conceptos más importantes de la Programación Orientada a Objetos. Por tanto, este libro surge con el objetivo de presentar los conceptos más importantes del paradigma de una manera sencilla y agradable al lector, apoyando los conceptos presentados con ejemplos que ayuden a la comprensión de todo lo expuesto.

Este libro se encuentra dividido en los siguientes ocho capítulos.

En el primer capítulo se contextualiza la Programación Orientada a Objetos dentro de la historia de la programación.

En el capítulo segundo se definen los conceptos básicos de clases y objetos junto con el principio básico de la Programación Orientada a Objetos: la encapsulación.

En el tercer capítulo se explican los mecanismos básicos de reutilización de código: la composición y la herencia.

En el capítulo cuatro se define el concepto de polimorfismo. Así, se explica el concepto de ligadura, las funciones virtuales, la sobrescritura, las clases abstractas, los constructores, funciones virtuales, destructores, y la diferencia entre sobrecarga y sobrescritura.

En el quinto capítulo se explica el manejo de errores de la programación tradicional y se compara con la foma de hacerlo en C++: manejo de excepciones.

En el sexto capítulo se define el concepto de estructura de datos y como ejemplo, se presenta una de las estructuras de datos más utilizadas: la estructura árbol.

Por su parte, en el capítulo siete, se presenta una metodología de identificación y modelado de clases de objetos.

Como último capítulo, en el capítulo ocho, se presentan otras características relevantes de C++ que se salen del ámbito de la programación orientada a objetos.

Finalmente, en el apéndice se pueden encontrar una relación de los errores más comunes, así como una batería de ejercicios para que el lector practique y comprenda todos los conceptos expuestos en este libro.

Índice general

1. Introducción **7**
1.1. Paradigmas de Programación: Evolución 7
1.1.1. Programación Estructurada 7
1.1.2. Programación Orientada a Objetos: Características base 8
1.1.3. Programación Orientada a Objetos: Historia 8
1.2. Procesos de Desarrollo Software 11
1.3. El lenguaje C++ 12
1.3.1. Lenguaje de ejemplo 13
1.4. Resumen 14

2. Clases y objetos **15**
2.1. Concepto de clase y objeto 15
2.1.1. Clases y objetos en C++ 16
2.1.2. El operador this 17
2.2. Encapsulación 17
2.2.1. Encapsulación en C++. Modificadores de acceso 18
2.3. Creación y eliminación de objetos 20
2.3.1. Constructores 20
2.3.1.1. Constructores en C++ 21
2.3.1.2. Clasificación de constructores 23
2.3.2. Destructores 24
2.3.2.1. Destructores en C++ 25
2.3.3. Ejemplo de creación y eliminación de objetos 25
2.3.4. Creación y eliminación dinámica de objetos 29
2.4. Paso de objetos como parámetros a funciones 30
2.4.1. Paso de objetos por valor 31
2.4.2. Paso de objetos por referencia 33
2.5. Funciones que devuelven objetos 35
2.5.1. Por referencia 35
2.5.2. Por valor 37
2.6. Miembros estáticos 37
2.6.1. Atributos miembro estáticos 37

2.6.2. Funciones miembro estáticas 38
2.7. Sobrecarga 39
2.7.1. Funciones sobrecargadas 40
2.7.2. Operadores sobrecargados 40
2.8. Resumen 43

3. Mecanismos de reutilización **45**
3.1. Composición 46
3.1.1. Problema 46
3.1.2. Concepto 48
3.1.3. La composición en C++ 48
3.1.4. Interfaz de la clase contenedora 50
3.2. Herencia 54
3.2.1. Problema 55
3.2.2. Concepto 55
3.2.3. Jerarquías de clases 56
3.2.4. La herencia en C++ 58
3.2.4.1. Control de acceso a los miembros de la clase base 58
3.2.4.2. Mayor control de acceso a los miembros de la clase base 63
3.2.4.3. Constructores y destructores en la herencia . 63
3.2.4.4. Redefinición 68
3.2.4.5. Conversiones entre objetos de la clase base y objetos de las clases derivadas 70
3.2.4.6. Herencia múltiple 71
3.2.4.7. Sobrecarga y redefinición 72
3.3. Resumen 73

4. Polimorfismo **75**
4.1. Concepto 75
4.2. Polimorfismo en C++ 76
4.2.1. Ligadura 77
4.2.2. Funciones virtuales 78
4.2.3. Sobrescritura 79
4.2.3.1. Extensibilidad y estructuras polimórficas . . . 79
4.2.3.2. Método virtual no sobrescrito 80
4.2.4. Clases abstractas 81
4.2.5. Constructores y funciones virtuales 82
4.2.6. Destructores 84
4.2.6.1. Destructor virtual 84
4.2.6.2. Destructor virtual puro 86
4.2.6.3. Destructores y funciones virtuales 86
4.3. Sobrecarga y sobrescritura 88
4.4. Resumen 90

5. Excepciones **91**
5.1. Concepto . . . 91
5.2. Manejo de errores tradicional . . . 91
5.3. Excepciones en C++ . . . 93
5.3.1. Lanzar y capturar excepciones . . . 93
5.3.2. Restricción de excepciones permitidas . . . 98
5.3.3. Relanzar excepciones . . . 99
5.3.4. Excepciones definidas por el usuario . . . 101
5.3.5. Excepciones en constructores y destructores . . . 102
5.4. Resumen . . . 104

6. Estructuras de datos **105**
6.1. Concepto . . . 105
6.2. Árboles . . . 106
6.2.1. Representación y recorrido de los árboles binarios . . . 107
6.2.2. Árbol binario de búsqueda . . . 108
6.2.3. Árboles balanceados . . . 109
6.2.4. Algoritmos principales de un ABO . . . 109
6.2.4.1. Inserción de nodos . . . 109
6.2.4.2. Eliminación de nodos . . . 110
6.2.4.3. Recorridos del árbol . . . 111
6.2.5. Implementación de un árbol binario ordenado . . . 112
6.3. Resumen . . . 119

7. Identificación y modelado **121**
7.1. Identificación . . . 122
7.1.1. Análisis gramatical para búsqueda de clases . . . 122
7.1.2. Clasificación de clases potenciales . . . 124
7.1.3. Clases obtenidas . . . 124
7.2. Identificación de atributos . . . 126
7.3. Identificación de operaciones . . . 128
7.3.1. Casos de Uso y Diagramas de Casos de Uso . . . 128
7.3.1.1. Diagrama de Casos de Uso de nuestro sistema 129
7.3.2. Análisis gramatical de verbos . . . 130
7.3.3. Estudio de las acciones . . . 131
7.3.4. Operaciones obtenidas . . . 131
7.4. Modelado del Diseño Estático . . . 134
7.4.1. Clases . . . 134
7.4.2. Relaciones . . . 136
7.4.3. Diagrama de Clases del Problema Propuesto . . . 141
7.5. Modelado de aspectos dinámicos . . . 143
7.5.1. Bifurcaciones e iteraciones . . . 145
7.5.2. Diagrama de Secuencias de una operación del Problema Propuesto . . . 147

7.6. Resumen 149

8. Otras características de C++ **151**
8.1. Programación genérica 151
8.1.1. Formato de plantilla de función 151
8.1.2. Plantillas de clase 153
8.2. Biblioteca estándar 156
8.2.1. Espacios de nombres 157
8.2.2. El tipo string 158
8.2.3. STL 160
8.3. Funciones friend 161
8.4. Funciones en línea 162
8.5. Resumen 163

A. Errores y ejercicios **165**
A.1. Relación de los errores más comunes 165
A.2. Ejercicios 166
A.2.1. Clases y objetos 167
A.2.2. Mecanismos de reutilización 168
A.2.3. Polimorfismo 169
A.2.4. Excepciones 169
A.2.5. Identificación y modelado 170

Capítulo 1

Introducción

1.1. Paradigmas de Programación: Evolución

El mundo del desarrollo de software está sometido a un proceso evolutivo constante. Actualmente, podemos ver cómo ha sufrido una gran evolución desde el código máquina y el ensamblador a través de conceptos como la programación procedural, la programación estructurada, la programación lógica, etcétera hasta llegar a la programación orientada a objetos, que es el paradigma más extendido hoy en día.

1.1.1. Programación Estructurada

Como surgió en su día el paradigma de Programación Estructurada para intentar paliar las deficiencias de la programación en ese momento, la Programación Orientada a Objetos aparece como una evolución natural en los paradigmas de programación que busca solucionar los principales problemas del paradigma anterior. Un breve resumen de las deficiencias que este nuevo paradigma intenta solventar sería el siguiente:

- Distinta abstracción del mundo. La programación clásica se centra en el comportamiento, normalmente representado por verbos, mientras que nuestra mente se centra en los seres, a los que, normalmente, identificamos con sustantivos.
- Dificultad en modificación y actualización. Los programas suelen tener datos compartidos por varios subprogramas, esto puede provocar que cualquier ligera modificación en un módulo afecte indirectamente al resto.
- Dificultad en mantenimiento. Es complicado encontrar todos los errores en grandes aplicaciones, lo que origina que muchos no aparezcan hasta estar la aplicación en explotación.

- Dificultad en reutilización. Normalmente las subrutinas son demasiado dependientes del contexto de un programa como para poder aprovecharlas en un nuevo programa.

A la hora de afrontar la construcción de grandes aplicaciones, estos defectos pueden provocar que muchos proyectos sean inabordables. La Programación Orientada a Objetos surgió con el objetivo de erradicar los problemas expuestos.

1.1.2. Programación Orientada a Objetos: Características base

La Programación Orientada a Objetos supone un cambio en la concepción del mundo de desarrollo de software, introduciendo un mayor nivel de abstracción que permite mejorar las características del código final. De manera muy básica, las aportaciones de este paradigma se pueden resumir en:

- Conceptos de clase y objeto, que proporcionan una abstracción del mundo centrada en los seres y no en los verbos.
- Los datos aparecen encapsulados dentro del concepto de clase. El acceso a los datos se produce de manera controlada e independiente de la representación final de los mismos. Como consecuencia, se facilita el mantenimiento y la evolución de los sistemas, al desaparecer las dependencias entre distintas partes del sistema.
- Mediante conceptos como la composición, herencia y polimorfismo se consigue simplificar el desarrollo de sistemas. La composición y la herencia nos permiten construir clases a partir de otras clases, aumentando en gran medida la reutilización.

La razón histórica del origen de estas aportaciones se explica en el siguiente apartado.

1.1.3. Programación Orientada a Objetos: Historia

Contrariamente a la creencia de mucha gente, la Programación Orientada a Objetos (POO a partir de ahora) no es un tema nuevo de discusión. Es cierto que en años recientes la POO ha tomado nueva fuerza y ha renacido como un paradigma. Sin embargo fue a finales de los años 60 cuando estas técnicas fueron concebidas. Para entender su origen hay que situarse en el contexto de la época, en el que el desarrollo y mantenimiento de proyectos software presentaban evidentes problemas de complejidad, coste y eficiencia. Uno de los grandes problemas que surgían consistía en la necesidad de adaptar el software a nuevos requisitos imposibles de haber sido planificados inicialmente. Este alto grado de planificación y previsión es contrario a la propia realidad. El hombre aprende y crea a través de la experimentación. La Orientación a Objetos brinda

estos métodos de experimentación, no exige la planificación de un proyecto por completo antes de escribir la primera línea de código.

Esto mismo pensaron en 1967, Krinsten Nygaard y Ole-Johan Dahl de la Universidad de Oslo, en el Centro Noruego de Computación, donde se dedicaban a desarrollar sistemas informáticos que realizaban simulaciones de sistemas mecánicos, por ejemplo motores, para analizar su rendimiento. En este desarrollo se encontraban con dos dificultades, por un lado los programas eran muy complejos y, por otro, forzosamente tenían que ser modificados constantemente. Este segundo punto era especialmente problemático; ya que la razón de ser de los programas era el cambio y no sólo se requerían varias iteraciones para obtener un producto con el rendimiento deseado, sino que muchas veces se querían obtener diversas alternativas viables, cada una con sus ventajas e inconvenientes.

La solución que idearon fue diseñar el programa paralelamente al objeto físico. Es decir, si el objeto físico tenía un número x de componentes, el programa también tendría x módulos, uno por cada pieza. Dividiendo el programa de esta manera, había una total correspondencia entre el sistema físico y el sistema informático. Así, cada pieza física tenía su abstracción informática en un módulo. De la misma manera que los sistemas físicos se comunican enviándose señales, los módulos informáticos se comunicarían enviándose mensajes. Para llevar a la práctica estas ideas, crearon un lenguaje llamado Simula 67.

Este enfoque resolvió los dos problemas planteados. Primero, ofrecía una forma natural de dividir un programa muy complejo y, en segundo lugar, el mantenimiento pasaba a ser controlable. El primer punto es obvio. Al dividir el programa en unidades informáticas paralelas a las físicas, la descomposición es automática. El segundo punto también se resuelve. En cada iteración de simulación, el analista querrá cambiar o bien piezas enteras o bien el comportamiento de alguna pieza. En ambos casos la localización de los cambios está perfectamente clara y su alcance se reduce a un componente, siempre y cuando la interfaz del mismo no cambie. Por ejemplo, si se estuviese simulando el motor de un coche, puede que se quisiera modificar el delco utilizado en la simulación anterior. Si el nuevo delco tuviera la misma interfaz (mismas entradas y salidas) o se cambiase sólo su comportamiento interno, nada del sistema (a parte del delco) estaría afectado por el cambio.

Con Simula 67 se introducen por primera vez los conceptos de clases, objetos, herencia, procedimientos virtuales y referencias a objetos (conceptos muy similares a los lenguajes Orientados a Objetos de hoy en día). En esta época, Algol 60 era el lenguaje de moda y Cobol el más extendido en aplicaciones empresariales, por lo que el nacimiento de Simula 67 y la Orientación a Objetos en Europa pasó inadvertido para gran parte de los programadores. En la actualidad Simula 67 se conoce simplemente como Simula, y contrariamente a lo que pudiera pensarse, todavía está en uso, mantenido por una pequeña comunidad de programadores, en su mayoría noruegos, y compiladores disponibles en la red.

El siguiente paso se da en los años 70 en los Estados Unidos, más concretamente en el Centro de Investigación de Palo Alto (PARC), California, donde Xerox tiene un centro de investigación en el que los científicos trabajan en conceptos que puedan convertirse en productos industriales al cabo de 10 a 20 años. En aquellos años contrataron a un joven llamado Alan Kay, que venía de la Universidad de Utah, donde había estado trabajando con gráficos, y había examinado un nuevo compilador procedente de Noruega, llamado Simula. Kay encontró conceptos, que más tarde aprovechó en Xerox, para llevar a término las ideas que proponía en su tesis doctoral. Éstas consistían básicamente en la propuesta de construcción de un ordenador llamado Dynabook, adecuado para ser utilizado por niños. El ordenador no tenía teclado, la pantalla era sensible al tacto y la mayor parte de la comunicación era gráfica. Al desarrollar este proyecto se inventaron el mouse y los entornos gráficos. Al volver a encontrarse en Palo Alto con una programación compleja y experimental, como en el caso de Nygaard y Dahl, Kay, junto con Adele Goldberg y Daniel Ingalls, decidieron crear un lenguaje llamado SmallTalk. Este lenguaje de programación es considerado el primer lenguaje Orientado a Objetos puro, dónde todo lo que se crea son clases y objetos, incluyendo las variables de tipos más simples. La primera versión se conoció como Smalltalk-72, aunque fueron surgiendo nuevas versiones (76,80,..) que pasaron de ser desarrolladas sólo para máquinas Xerox a serlo para la mayoría de plataformas disponibles.

Hasta este momento, uno de los defectos más graves de la programación era que las variables eran visibles desde cualquier parte del código y podían ser modificadas incluyendo la posibilidad de cambiar su contenido (no existen niveles de usuarios o de seguridad, o lo que se conoce como visibilidad). D. Parnas fue quien propuso la disciplina de ocultar la información. Su idea era encapsular cada una de las variables globales de la aplicación en un sólo módulo junto con sus operaciones asociadas, de forma que sólo se podía tener acceso a estas variables a través de sus operaciones asociadas. El resto de los módulos (objetos) podían acceder a las variables sólo de forma indirecta mediante las operaciones diseñadas para tal efecto.

En los años ochenta surgen nuevos lenguajes orientados a objetos basados en Lisp. Esto se debe principalmente a que este lenguaje disponía de mecanismos que permitían la implementación basándose en orientación a objetos. Entre los lenguajes surgidos de Lisp podemos destacar Flavors, desarrollado en el MIT, Ceyx, desarrollado en el INRIA, y Loops, desarrollado por Xerox. Este último introdujo el concepto de la programación orientada a datos con la que se puede enlazar un elemento de datos con una rutina y que dio paso a la programación controlada por eventos. La mayoría de las vertientes tomadas por estos lenguajes se fusionaron en CLOS (Common Lisp Object System) que fue el primer lenguaje orientado a objetos que tuvo un estándar ANSI. En 1984, Bjarne Stroustrup de ATT-Bell se planteó crear un sucesor al lenguaje C, e incorporó las principales ideas de Smalltalk y de Simula 67, creando el lenguaje C++, quizás el lenguaje responsable de la gran extensión de los conceptos de

la orientación a objetos. Posteriores mejoras en herramientas y lanzamientos comerciales de C++ por distintos fabricantes, justificaron la mayor atención hacia la programación Orientada a Objetos en la comunidad de desarrollo de software. El desarrollo técnico del hardware y su disminución del costo fue el detonante final. En esta misma década se desarrollaron otros lenguajes Orientados a Objetos como Objective C, Object Pascal, Ada y otros.

En el inicio de los años 90 se consolida la Orientación a Objetos como una de las mejores maneras para resolver problemas de programación. Aumenta la necesidad de generar prototipos más rápidamente (concepto RAD Rapid Aplication Development) sin esperar a que los requisitos iniciales estén totalmente precisos. En esta década se implanta con fuerza el concepto de reutilización, cuyo sentido es facilitar la adaptación de objetos ya creados a otros usos diferentes a los originales sin necesidad de modificar el código ya existente. A medio y largo plazo, el beneficio que puede obtenerse derivado de la reutilización es muy importante, y, quizás, es la razón principal por la que la industria informática se ha abocado a la orientación a objetos. Avanzando algunas cifras, se puede indicar que los niveles de reutilización de software pasan del 5-15 % en centros no orientados a objetos, a niveles por encima del 80 % en los orientados a objetos.

Con la reutilización como pilar básico de su filosofía, un equipo de Sun Microsystems crea Java en 1995, con el objetivo de conquistar el mundo de la programación, teniendo gran éxito en aplicaciones en red. No en vano, captó mucha atención en los primeros meses de 1996 en base a ser considerado el medio para dominar Internet. Java es totalmente orientado a objetos. Está basado en C++, intentando evitar los principales problemas del mismo, como, por ejemplo, el acceso directo a memoria dinámica. Se fundamenta en el uso de bytecode (código de bajo nivel, portable e interpretable) y una máquina virtual accesible para todo el mundo que ejecuta esos bytecodes. Hoy en día es lo más cercano a lo que podría denominarse una máquina universal.

El mundo de los lenguajes de programación orientados a objeto evoluciona día a día y, continuamente, surgen nuevos lenguajes o nuevas versiones de lenguajes ya consolidados.

1.2. Procesos de Desarrollo Software

Evidentemente, el desarrollo de un sistema software debe seguir un proceso que especifique cómo debe ser, que permita controlar y seguir la evolución del sistema y que permita establecer aproximadamente su coste. El estudio de las características de estos procesos se lleva a cabo en el ámbito de la comunidad de Ingeniería del Software.

Básicamente, podemos hablar de dos tipos procesos:

- En cascada.

- Iterativo (o incremental).

La diferencia fundamental entre ellos consiste en la forma de dividir un proyecto en partes más pequeñas.

Los procesos en cascada dividen un proyecto según las actividades. La construcción de un sistema software conlleva una serie de actividades que componen el ciclo de vida software: análisis de requisitos, diseño, implementación y pruebas. De esta manera, los procesos en cascada dividirán un proyecto de 1 año de duración en, por ejemplo, una fase de análisis de 2 meses, una fase de diseño de 4 meses, una fase de implementación de 3 meses y una de prueba de 3 meses.

Por su parte, los procesos iterativos dividen un proyecto en subconjuntos de funcionalidad. De esta forma, un proyecto de 1 año se dividiría en 3 iteraciones de 4 meses. En la primera iteración, se realizaría todo el ciclo de vida software para la cuarta parte de los requisitos. El resultado de la primera iteración sería un sistema software con la cuarta parte de la funcionalidad. En las siguientes iteraciones se continuaría evolucionando ese sistema, introduciendo el resto de funcionalidad.

Dentro de la comunidad orientada a objetos, se suele optar por procesos iterativos que responden mejor al problema derivado de los cambios en los requisitos. En muchos campos de aplicación, los requisitos van cambiando a lo largo del desarrollo del sistema. Por lo tanto, los procesos en cascada se muestran menos efectivos; ya que, su flexibilidad ante estos cambios es mucho menor.

1.3. El lenguaje C++

Como ya vimos anteriormente, C++ nace a mediados de los ochenta de la mano de Bjarne Strouptup. En las primeras versiones era un simple preprocesador que convertía las modificaciones añadidas, clases y tratamiento riguroso de tipos, a lenguaje C. De hecho, el nombre de C++ le viene en parte por ser considerado una evolución natural del lenguaje C. En la actualidad, C++ provee mecanismos suficientes para implementar, entre otras cosas, todos los conceptos de la POO de manera adecuada y, por supuesto, tiene su propio compilador nativo, sin que ello sea obstáculo para que, la mayoría de los compiladores de C++ existentes, también compilen C puro. Quizás sea este soporte de múltiples paradigmas una de las claves de su éxito.

Algunas de las características más importantes de C++ son:

- Soporte de diferentes estilos de programación.
- Espacios de nombres.
- Encapsulación de información mediante el concepto de clases, apoyado también en los modificadores de acceso.

- Soporte de herencia simple y múltiple.
- Funciones virtuales, que posibilitan el polimorfismo.
- Sobrecarga de funciones y operadores.
- Soporte de excepciones.
- Biblioteca estándar, base para el desarrollo de un gran número de otras bibliotecas.
- Programación genérica mediante el soporte de plantillas.

Como se puede apreciar, el lenguaje C++ va más allá del paradigma de programación orientada a objetos. Sin embargo, en este libro nos vamos a centrar casi exclusivamente en los conceptos y abstracciones propuestos por este paradigma. El fin último de este libro es introducir al lector en la programación orientada a objetos. Este no es el libro adecuado para el lector interesado en aprender en profundidad toda la potencia del lenguaje C++.

1.3.1. Lenguaje de ejemplo

El lenguaje C++ ha sido elegido por los autores de este libro para plasmar de forma práctica los conceptos de programación orientada a objetos. Si bien es verdad que existen lenguajes orientados a objetos más modernos y más ajustados al paradigma de POO, hemos elegido C++ por las siguientes razones:

- Su gran expansión. Se trata de un lenguaje mundialmente aceptado, a pesar de sus características más peculiares.
- Sometido a un estándar. El estándar C++ (ISO/IEC 14882) se ratificó en 1998. Gracias al proceso de estandarización que ha sufrido y a la biblioteca estándar C++, este lenguaje se ha convertido en un lenguaje potente, eficiente y seguro.
- Existe un gran número de compiladores y de entornos de desarrollo. Es un lenguaje que puede ser usado en diferentes plataformas y diversos dominios de aplicación. Además, posee un gran número de herramientas de desarrollo.
- Demanda profesional. Es un lenguaje muy demandado profesionalmente, sobre todo, en dominios en los que el rendimiento es un requisito fundamental.

1.4. Resumen

Este capítulo pretende situar al lector en el contexto que dio origen al paradigma de programación orientada a objetos. El objetivo consiste en proporcionar una dimensión histórica a los conceptos introducidos por este paradigma, de manera que se pueda lograr una comprensión global de cada uno de ellos. En definitiva, este capítulo indaga sobre las causas que motivaron el origen de estos conceptos, de esta nueva forma de representación de la realidad.

Por otra parte, se introduce el lenguaje C++ que va a servir de herramienta para llevar a la práctica todos los conceptos de programación que presenta este libro.

Capítulo 2

Clases y objetos

2.1. Concepto de clase y objeto

El mundo está lleno de objetos: el coche, la lavadora, la mesa, el teléfono, etc. El paradigma de programación orientada a objetos proporciona las abstracciones necesarias para poder desarrollar sistemas software de una forma más cercana a esta percepción del mundo real.

Mediante la POO, a la hora de tratar un problema, podemos descomponerlo en subgrupos de partes relacionadas. Estos subgrupos pueden traducirse en unidades autocontenidas llamadas **objetos**. Antes de la creación de un objeto, se debe definir en primer lugar su formato general, su plantilla, que recibe el nombre de **clase**. Por ejemplo, pensemos que estamos intentando construir una aplicación de dibujo de cuadrados. Nuestro programa va a ser capaz de pintar cuadrados con diferentes tamaños y colores: uno será rojo y de 3 centímetros de lado, otro verde y de 5 centímetros de lado, etc. Como podemos deducir, cada cuadrado está definido por dos características, *lado* y *color*, y contiene la operación *dibuja*. Desde el punto de vista de la POO, cada cuadrado se va a representar como un objeto de la clase *Cuadrado* que es la contiene las características y operaciones comunes de los objetos. En realidad, estamos creando un nuevo tipo, el tipo *Cuadrado*, y cada variable de este tipo especial recibe el nombre de objeto.

Una clase, por lo tanto, puede definirse como un tipo de datos cuyas variables son objetos o instancias de una clase. Puede decirse que el concepto de clase es estático y el concepto de objeto es dinámico; ya que, sólo existe en tiempo de ejecución. Cada objeto de una clase comparte con el resto las operaciones, y a la vez posee sus propios valores para los atributos que posee: esto es, su **estado**. Las operaciones, también llamadas **mensajes**, se usarán tanto para cambiar el estado de un objeto como para comunicar objetos entre sí mediante el paso de mensajes.

Un ejemplo de clase podría ser la clase *Alumno*, esta clase definiría diferentes alumnos, en términos de *nombre*, *número de matrícula*, *curso*, *especialidad*, *localidad*, etc... el alumno Juan Sánchez con número de matrícula 5566432 sería una instancia de la clase *Alumno*. Otro ejemplo de clase podría ser la clase *Telefono*, esta clase definiría diferentes teléfonos con los datos número, tipo de tarifa asociada, nombre de usuario asociado, e importe acumulado por cada teléfono. Una instancia de esta clase podría ser el teléfono con número 927123456.

2.1.1. Clases y objetos en C++

La sintaxis de declaración de una clase en el lenguaje C++ es muy parecida a la de las estructuras.

```
class nombre de clase { cuerpo de la clase };
```

El cuerpo de la clase está compuesto por los datos o **atributos miembro** y por las operaciones o **funciones miembro**.

Veamos cómo se implementaría en C++, siguiendo los principios de la POO, la clase *Cuadrado* de la aplicación de dibujo de cuadrados.

```
enum Color {rojo,verde,azul,amarillo,blanco,negro};
class Cuadrado {
  int lado;
  Color color;
public:
  void dibuja( );
};
```

Como se puede ver en el ejemplo, la clase *Cuadrado* contiene los atributos miembro que caracterizan cada instancia de cuadrado y una función miembro que nos permite dibujar cada una de los objetos cuadrado.

En el ejemplo anterior, sólo se ha presentado la declaración de la clase *Cuadrado*. Para completar ese ejemplo, falta definir la función miembro *dibuja*. En C++, el formato de definición de una función miembro es el siguiente:

```
tipo nombre clase::nombre función (lista de parámetros) {
  //cuerpo de la función miembro
}
```

Como se puede ver en la definición de una función miembro, la definición debe incluir el nombre de la clase, esto se debe a que podría haber dos o más clases que tuvieran funciones miembro con el mismo nombre. El operador "::" utilizado se llama operador de resolución de alcance y se utiliza para indicar a qué clase pertenece la función miembro. Nuestro ejemplo se completaría entonces de la siguiente manera.

```
void Cuadrado::dibuja( ){
//Algoritmo para dibujar un cuadrado
}
```

Cuando se definen varios objetos de una misma clase, todos ellos comparten las funciones miembro definidas en la clase. Los objetos se diferencian por los valores que toman sus atributos. A este conjunto de atributos que distinguen a unos objetos de otros se les llama estado del objeto. Por ejemplo, todos los cuadrado comparten la misma definición de la función *dibuja*, pero poseen diferentes valores de sus atributos miembro *lado* y *color*.

2.1.2. El operador this

La palabra clave **this** puede utilizarse en cualquier método de la clase para referirse al objeto actual. Cada vez que se llama a una función miembro, automáticamente se pasa un puntero al objeto que la llama. Se puede acceder a este puntero utilizando **this**.

Veamos el uso del operador this en el ejemplo anterior.

```
class Cuadrado {
  int lado;
  Color color;
public:
  void dibuja( );
  void modifica(int lado, Color color){
    this->lado = lado;
    this->color = color;
  }
};
```

Hemos introducido una nueva función miembro a la clase *Cuadrado*: la función *modifica*. En el cuerpo de esta función estamos usando el operador **this** para distinguir, en este caso, los atributos de los parámetros de la función *modifica*; ya que tienen el mismo nombre. Este ejemplo sencillo sólo quiere poner de manifiesto la posibilidad de acceder al objeto actual a través del puntero **this**. En el caso de no utilizarse en este ejemplo el puntero **this**, la modificación no se habría producido en los atributos *lado* y *color*.

2.2. Encapsulación

Una de los pilares básicos de la POO es la encapsulación de los datos. Según los principios de este paradigma de programación, el acceso a los datos de una clase debe realizarse de forma controlada, protegiéndolos de accesos no deseados. Cuando se desarrolla una aplicación, a veces es necesario ocultar los

tipos de datos usados para que el usuario permanezca independiente de los detalles de los mismos. De esta manera, el usuario no es sensible a los cambios que se puedan producir en los tipos de datos elegidos dentro de una clase.

La encapsulación en los lenguajes orientados a objeto suele lograrse al declarar algunos datos como privados. El acceso a estos datos sería siempre controlado; ya que, se haría siempre a través de funciones miembro que realizarían las modificaciones oportunas de una forma transparente al usuario.

La encapsulación es el mecanismo que enlaza el código y los datos, a la vez que los asegura frente a accesos no deseados. La principal razón del uso de la encapsulación es evitar el acceso directo a atributos de una clase desde fuera de la propia clase. Se busca, principalmente, que el acceso a estos atributos se realice siempre mediante funciones miembro de la propia clase. El acceso a ciertos elementos de datos se puede controlar de forma estricta considerándolos privados.

Una vez presentado el concepto encapsulación, se puede definir de nuevo un objeto como una entidad lógica que encapsula los datos y el código que manipula dichos datos. Dentro de un objeto, parte del código y/o los datos puede ser privada al objeto y, por tanto, inaccesible desde fuera del mismo. Se podrá asegurar un acceso correcto al objeto utilizando las funciones miembro del objeto para acceder a los datos privados. El acceso a los datos de una clase se establece por lo tanto a través de los métodos que implementa la propia clase y que pone a disposición del usuario. Este mecanismo recibe el nombre de paso de mensajes o interfaz y es la forma de comunicación que se establece entre objetos.

2.2.1. Encapsulación en C++. Modificadores de acceso

Una clase puede contener tanto partes privadas como partes públicas. Por defecto, todos los elementos de la clase son privados, lo que significa que no son accesibles desde ninguna función que no sea miembro de la clase. El mecanismo aportado por C++ para conseguir la encapsulación es la utilización de los modificadores de acceso **private** y **public**. Los miembros privados ocultan las estructuras de datos y los procedimientos internos de un objeto.

Si se incluye el modificador de acceso **private** en la lista de atributos y funciones miembro, todos los miembros que aparezcan a continuación serán miembros privados. Los atributos serán atributos miembro privados y las funciones serán funciones miembro privadas.

Por su parte, para hacer públicas ciertas partes de una clase, se deben declarar a continuación del modificador de acceso **public**. Aunque el lenguaje C++ permite tener atributos **public**, debe restringirse su utilización. En su lugar, por norma general, para lograr la encapsulación de los datos, es aconsejable crear todos los datos privados y proporcionar acceso a los mismos mediante funciones públicas. Introduciendo los modificadores de acceso, el formato de declaración de una clase quedaría así:

```
class nombre de clase {
   private:
     //miembros privados
   public:
     //miembros públicos
};
```

Una clase podrá tener el número que desee de secciones de cada tipo, aunque lo normal será una sección de cada tipo. La cláusula **private** mostrada en el formato de declaración de una clase es opcional, ya que los miembros que aparecen después de la llave de apertura hasta la aparición de una cláusula distinta se consideran siempre miembros privados.

Por un lado, las funciones que son públicas o accesibles para otras clases forman el interfaz de una clase y son declaradas como **public**. Por otro lado, las funciones que no permiten el acceso desde otras clases, es decir, aquellas cuyo uso es de tipo interno a la clase en la que están definidas son declaradas como **private**. Se puede decir que estas funciones están ocultas para el resto de clases.

A continuación, se introduce un nuevo ejemplo para mostrar el funcionamiento de los modificadores de acceso. La declaración de la clase *Telefono* es la que se muestra a continuación.

```
class Telefono {
  string numero;
  string tipo_tarifa;
  string usuario;
  float importe;
  void inicializarImporte( );
public:
  void actualizarImporte(float _importe);
  float obtenerImporte( );
  void visualizarUsuario( );
};
//definición
void Telefono::inicializarImporte() {
  importe=0;
}
void Telefono::actualizarImporte(float _importe) {
  if (_importe<0)
    inicializarImporte();
  else
    importe+= _importe;
}
float Telefono::obtenerImporte( ) {
```

```
  return importe;
}
void Telefono::visualizarUsuario( ) {
  cout << "El usuario asociado es: " << usuario <<endl;
}
void main(){
  Telefono telefono;

  cout<< "El número de teléfono es";
  cout<< telefono.numero <<endl; //error
  telefono.inicializarImporte(); //error
  telefono.actualizarImporte(5);
  float x = telefono.obtenerImporte();
  telefono.visualizarUsuario();
}
```

En este ejemplo, se ha optado por declarar todos los atributos miembro y la función *inicializarImporte* como privados. Obsérvese que no es necesario incluir el modificador **private**. Mientras que el resto de las funciones miembro se declaran como **public**, especificando el interfaz de la clase *Telefono*. Como se puede apreciar en la función **main**, el compilador no permitirá el acceso a los miembros privados del objeto *telefono*.

2.3. Creación y eliminación de objetos

La creación y eliminación de objetos se realiza de la misma manera que para el resto de las variables de un programa. Normalmente, se crea cuando se declara y se destruye cuando termina su ámbito. Por ejemplo, si se declara un objeto *telefono* dentro de una función, se comportará como una variable local, es decir, se creará cada vez que la ejecución del programa entre en dicha función y se destruirá cuando se llegue al final de la misma.

La única peculiaridad de los objetos consiste en que cuando se crean se invoca siempre a una función especial denominada **constructor**, mientras que cuando se destruyen se invoca siempre a una función especial denominada **destructor**. Éste suele ser el comportamiento general de los lenguajes orientados a objeto.

2.3.1. Constructores

Normalmente necesitamos inicializar alguno de los atributos miembro de un objeto, o todos, cuando se declara el objeto. Para realizar la inicialización de los atributos, al definir una clase se utiliza un tipo especial de función miembro, ésta se llama **constructor**. Cuando se crea un objeto de una clase, se ejecutará automáticamente la función miembro constructor. El constructor se utilizará

por lo tanto para inicializar los atributos miembro que se deseen de un objeto y para realizar algún otro tipo de inicialización que el usuario crea necesaria.

2.3.1.1. Constructores en C++

La definición de un constructor se realiza como cualquier otra función miembro, teniendo en cuenta dos particularidades: La función constructor debe tener el mismo nombre que la clase. Por ejemplo, en la clase *Telefono*, su constructor sólo puede llamarse *Telefono*. Además, la función constructor no puede devolver ningún tipo de datos, incluyendo el tipo **void**.

La sintaxis general de un constructor es la siguiente:

```
nombre de clase::nombre de clase (lista de parámetros) {
  //inicialización de datos miembro del objeto
}
```

Los constructores deben definirse siempre de tipo **public**; ya que, de otra forma, no podríamos declarar ningún objeto como instancia de esa clase, al no poder invocarse a su constructor.

Para declarar e inicializar objetos de la clase *Telefono* se escribe el nombre de la clase (nombre del tipo) y el nombre del nuevo objeto a declarar, como si se tratara de otro tipo de variable, de la forma que aparece a continuación:

```
Telefono telefono1;
```

De este modo, cuando se crea el objeto *telefono1* de la clase *Telefono*, el compilador llama automáticamente al constructor de la clase y se inicializan las variables de dicho objeto.

Como se muestra en el formato de un constructor, éste se define de la misma forma que cualquier otra función miembro, anteponiendo el nombre de la clase, escribiendo "::" y a continuación el nombre del constructor, que es igual al nombre de la clase. Teniendo en cuenta además que el constructor no puede devolver ningún tipo de datos.

En el ejemplo de la clase Telefono, la definición de su constructor podría ser de la siguiente forma:

```
Telefono::Telefono() { importe=0; }
```

Este constructor podría utilizarse por ejemplo para inicializar el *importe* de la clase *Telefono* al valor cero.

A veces es necesario inicializar los objetos de distinta forma. Por ejemplo, podría interesarnos que al crear un objeto sin parámetros se inicializaran las variables con ciertos valores y al crearlo con parámetros las variables tomaran los valores de esos parámetros de entrada. En este caso, deberíamos crear un

nuevo constructor, implementando otra función que recibiera parámetros de entrada. En nuestro ejemplo podríamos definir un constructor que no recibiera parámetros de entrada e inicializara el dato importe a cero y otro que recibiera tres parámetros e inicializara los datos con dichos parámetros. Realmente, nos estamos aprovechando de la capacidad de C++ para sobrecargar funciones. Más adelante veremos qué significa la sobrecarga de funciones.

```
class Telefono {
  string numero;
  string tipo_tarifa;
  string usuario;
  float importe;
public:
  Telefono();
  Telefono (string _numero, string _tarifa, string _usuario);
  void actualizarImporte(float _importe);
  float obtenerImporte( );
  void visualizarUsuario( );
};
//definición
void Telefono::Telefono() { importe=0; }
void Telefono::Telefono(string _numero,
                        string _tarifa,
                        string _usuario) {
  importe = 0;
  numero = _numero;
  tarifa = _tarifa;
  usuario = _usuario;
}
```

Un constructor que no recibe parámetros se llama constructor **predeterminado**. El compilador de C++ genera automáticamente este tipo de constructor cuando no incluimos ningún constructor en la declaración de la clase, este constructor se encarga de realizar una inicialización bit a bit de los atributos del objeto. Por lo tanto, aunque no definamos ningún constructor, hemos de ser conscientes de que todas las clases tendrán una función miembro constructor por defecto. Por su parte, un constructor que recibe parámetros de entrada se llama constructor **parametrizado**. La definición de ambos constructores ha sido mostrada en el ejemplo anterior, donde se ha definido la clase *Telefono*.

Las distintas formas de crear objetos utilizando los diferentes constructores definidos para la clase son las siguiente:

```
<nombre de clase> objeto;    //por defecto
<nombre de clase> objeto (parámetros); //parametrizado
```

Siguiendo nuestro ejemplo, podríamos crear objetos de tipo *Telefono* utilizando los constructores definidos anteriormente.

```
Telefono telefono1;
Telefono telefono2("927123456", "Di", "Juan Sánchez");
```

El objeto *telefono1* ha sido creado con el constructor por defecto, con lo cual sólo se inicializa la variable miembro importe con valor cero, sin embargo, el objeto *telefono2* ha sido creado con el constructor parametrizado, de esta forma las variables miembro se han inicializado con los valores introducidos por parámetro en la creación del objeto.

Se pueden crear también arrays de objetos, de la misma forma que se declaran arrays de otro tipo de variables:

```
Telefono telefonoarray[6];
```

Como se puede ver en el ejemplo, al crear el array de objetos no se pasan parámetros. En este caso se estaría produciendo una llamada al constructor por defecto.

Como se puede observar, es útil la definición de un constructor por defecto que realice las operaciones que creamos oportunas cuando se crean nuevos objetos de forma que tengamos valores controlados de los atributos de estos nuevos. Además, tiene sentido que el compilador sintetice el constructor por defecto en caso de que no lo haya hecho el programador.

2.3.1.2. Clasificación de constructores

Se pueden diferenciar diferentes tipos de constructores de objetos:

- Constructor **predeterminado**, por defecto o por omisión. Es aquel constructor que no requiere ningún argumento. Se aplica en el caso predeterminado donde se declara un objeto sin especificar argumentos. Si no incluimos ningún constructor en una clase, el compilador generará un constructor predeterminado que inicializará los atributos miembros con los valores por defecto. Este constructor se invocará al declarar objetos de la clase. En el ejemplo, el constructor predeterminado será:

  ```
  Telefono ();
  ```

- Constructor **parametrizado** u ordinario. Crea una nueva instancia de la clase e inicializa los datos miembro con los valores incluidos en la declaración. Pueden existir distintos constructores de este tipo, dependiendo del número de parámetros que incluyan en su definición. En nuestro ejemplo, un constructor parametrizado podría ser de la siguiente forma:

  ```
  Telefono (string _numero,string _tarifa,string _usuario);
  ```

- Constructor **copia**. Se puede definir como un caso especial de constructor parametrizado. Recibe como parámetro una referencia a un objeto de la misma clase. Se utiliza este constructor cuando utilizamos un objeto de una clase para inicializar otro objeto de la misma clase. Cuando creamos un nuevo objeto utilizando un constructor copia, se crea este objeto en otra zona de memoria, es decir se hace una copia física del objeto. Este constructor tiene tanta importancia dentro del lenguaje C++ que si no lo define el programador, el compilador se encarga de sintetizar uno automáticamente que realiza la copia bit a bit del objeto. El formato del constructor copia es de la siguiente forma:

  ```
  nombre_de_clase (const nombre_de_clase &objetonuevo);
  ```

  Este constructor es invocado automáticamente cuando es necesario crear una copia de un objeto, por ejemplo, en el paso de objetos como parámetros por valor. Este caso constituye la fuente de muchos errores entre los programadores inexpertos de C++ y será considerado con más detenimiento un poco más adelante. A parte de este caso, también se invocará cuando creamos objetos de esta forma:

  ```
  Telefono telefono1; //constructor por defecto
  Telefono telefono2 = telefono1; //constructor copia
  telefono3(telefono1); //constructor copia
  ```

  El operador = en este caso no señala una asignación, sino una invocación al constructor copia; ya que, se está usando para declarar un nuevo objeto a partir de otro. Siguiendo con el ejemplo, para el caso de nuestra clase *Telefono*, el constructor copia sería:

  ```
  Telefono (const Telefono &tel);
  ```

2.3.2. Destructores

El complemento de un constructor es el **destructor**. Al igual que a veces es necesario realizar inicializaciones u otro tipo de acciones al crear un objeto, a veces necesitamos realizar ciertas acciones cuando el objeto se destruye. Una de las ocasiones en las que se hace necesario el uso de un destructor es cuando se necesita liberar los recursos previamente reservados.

Algunos lenguajes orientados a objetos no definen explícitamente destructores para sus objetos, sino que utilizan otros mecanismos para la liberación de los recursos consumidos por los objetos. Un conocido mecanismo de esta índole es el recolector de basura, que se encarga de liberar los recursos de forma automática y transparente al programador. La discusión asociada a la inclusión o no de mecanismos de este tipo dentro del lenguaje queda fuera del alcance

de este libro, pero el lector interesado encontrará referencias sobre este tema dentro de la bibliografía.

2.3.2.1. Destructores en C++

La forma de declarar un destructor es igual a la forma de declarar un constructor con la excepción de que va precedido del símbolo "~" y que no puede recibir parámetros de entrada.

Los métodos constructores y destructores no se invocan específicamente desde el programa, es el compilador el que realiza dicha invocación. En el caso del constructor se realiza cuando se crea un objeto y en caso del destructor cuando finaliza el ámbito de un objeto, es decir, cuando se sale de la función donde ha sido creado el objeto o cuando se elimina explícitamente dicho objeto.

En nuestro ejemplo, el destructor para la clase *Telefono* se declararía de la siguiente forma:

```
~ Telefono ();
```

En la definición del destructor, se liberarían los recursos que se hubieran reservado a la hora de crear un objeto de la clase *Telefono*.

Para finalizar con el concepto de destructor, es necesario volver a recordar que cuando finaliza el ámbito de los objetos creados se invocará automáticamente al destructor de cada uno de ellos, realizando las operaciones definidas en él, normalmente, liberando recursos. Por lo tanto, es muy importante controlar en todo momento cuándo se van a crear y destruir los objetos de nuestra aplicación, para evitar intentar acceder a un objeto que haya sido destruido.

2.3.3. Ejemplo de creación y eliminación de objetos

El siguiente ejemplo, que parte de una reducción de la clase *Telefono*, trata de mostrar cómo se producen las distintas invocaciones de los constructores y destructores al trabajar con objetos, haciendo hincapié en el peligro de no definir un constructor copia para algunas clases. Antes de comenzar, es necesario comentar el hecho de que este ejemplo usa cadenas al modo C para reflejar el problema de no definir los constructores copia en ciertos casos. Se podría haber usado un puntero a un objeto de una clase creada para este ejemplo, pero se ha usado el puntero al tipo **char** para reducir el tamaño del ejemplo. No obstante, reseñar que el problema que se va a exponer se produce por el hecho de usar atributos miembro de una clase que reserven memoria dinámica, sin importar de qué tipo o clase sean los punteros. Finalmente, recomendar al lector el uso de cadenas al estilo ANSI C++ mediante la clase **string** que forma parte de la librería estándar de C++ (esta librería se verá en el apéndice de este libro).

```
class Telefono {
  char *numero;
```

```
  float importe;
public:
  Telefono (); //constructor por defecto
  Telefono (char *_numero); //constructor parametrizado
  void mostrar();
  ~ Telefono (); //destructor
};
Telefono::Telefono () {
  numero = new char[4];
  strcpy(numero,"555");
  importe = 0;
  cout <<"Constructor por defecto"<<endl;
}
Telefono::Telefono (char *_numero) {
  numero=new char[strlen(_numero)+1];
  assert(numero); //finaliza el programa si error
  strcpy(numero, _numero);
  importe=0;
  cout <<"Constructor parametrizado"<<endl;
}
void Telefono::mostrar() {
  cout<<"Número :"<<numero<<endl;
}
Telefono::~Telefono () {
  strcpy(numero,"xxx"); //Ponemos 3 x en la memoria
                        //a la que apunta numero
  delete numero;
  cout <<"Destructor"<<endl;
}
void main() {
  Telefono telefono1;
  Telefono telefono2("927123456");
  Telefono *telefono3 = new Telefono(telefono2);
  telefono1.mostrar();
  telefono2.mostrar();
  telefono3->mostrar();
  delete telefono3; //borramos la copia
  telefono1.mostrar();
  telefono2.mostrar();
}
```

Hemos creado la clase *Telefono*, definiendo el constructor por defecto, el destructor y un constructor parametrizado. Esta clase posee un atributo miembro dinámico, *numero*, que es un puntero a carácter. A este atributo es necesario

asignarle espacio en el montón (heap) cuando se crea un objeto de esta clase; por lo tanto, se le asigna espacio en los constructores. Además, cuando se destruya un objeto se debe liberar la memoria que ocupa el atributo numero de ese objeto. Por esta razón, se invoca a **delete** sobre ese puntero en el destructor. En resumen, en nuestra implementación hemos tenido cuidado con la gestión de memoria para este atributo. Por otro lado, en la función main se crean tres objetos de la clase *Telefono*. El primero mediante el constructor por defecto, el segundo mediante el constructor parametrizado definido y el tercero se crea dinámicamente como una copia del segundo, en el siguiente apartado se explica con más detalle la creación y eliminación dinámica de objetos. Observemos que no se ha definido el constructor copia. Veamos cuál es la salida de este ejemplo.

```
Constructor por defecto
Constructor parametrizado
Número: 555
Número: 927123456
Número: 927123456
Destructor
Número: 555
Número: xxx
Destructor
Destructor
```

Lo primero que aparecen son las dos llamadas a los constructores que corresponden a los dos primeros objetos *Telefono* que se crean. Después se muestra el contenido de cada uno de los objetos. El contenido del objeto *telefono3* coincide con el del *telefono2*, ya que se trata de una copia del mismo. El problema surge cuando destruimos el objeto *telefono3*. Observemos que el destructor pone la cadena "xxx" en el espacio de memoria al que apunta el atributo numero de cada objeto antes de liberar la memoria ocupada. Como podemos observar, al invocar a la función *mostrar* sobre el objeto *telefono2* aparece la cadena "xxx" en lugar de "927123456", que era el valor que debería tener el atributo *numero* del objeto *telefono2*. ¿Por qué ha variado el valor del atributo *numero* del objeto *telefono2*? Viendo el código, podemos concluir que algo ha pasado con este atributo cuando hemos destruido el objeto *telefono3*. Volvamos a repasar cómo se ha creado el objeto *telefono3*. Hemos dicho que era una copia del objeto *telefono2* por lo que se ha debido crear a través del constructor copia de la clase *Telefono*. Sin embargo, no hemos definido ningún constructor copia en esta clase, por lo que el compilador ha sintetizado uno que realiza una copia bit a bit. La copia bit a bit significa que para cada atributo se va a copiar el valor que almacena a nivel de bit sin tener en cuenta el tipo del atributo. En este caso, para el atributo *numero* se va a copiar la dirección de memoria que contiene, que apuntará a la zona de memoria dinámica donde se guarda el valor de la cadena *numero*. Por lo tanto, el atributo *numero* del objeto *telefono3* toma el

valor de la dirección de memoria que guarda el valor de *numero* de *telefono2*. En conclusión, los atributos *numero* de ambos objetos apuntan a la misma posición de memoria, que es donde se guarda la cadena "927123456". De esta forma, al mostrar el *numero* de *telefono3* parece que todo funciona correctamente. Como ya intuíamos, el problema surge al destruir *telefono3*. Al destruir este objeto, se libera la memoria que ocupa su atributo *numero*. El problema es que esa memoria era la misma a la que apunta el *numero* de *telefono2*. Por lo tanto, al destruir el *telefono3*, estamos destruyendo el valor de dicho atributo de *telefono2*. La figura 2.1 expresa gráficamente este comportamiento erróneo, mostrando en colores la memoria reservada y la liberada.

El problema surge porque la copia bit a bit no es capaz de crear copias de atributos que se encuentran en memoria dinámica, sino sólo de los punteros que la direccionan. Por lo tanto, para evitar errores de este tipo es necesario definir el constructor copia siempre que vayamos a necesitar copias de un objeto. Debemos ser conscientes de que en el paso de objetos como parámetros por valor o el retorno de objetos por valor, el compilador crea una copia de los objetos. Este punto se comenta más adelante porque suele ser fuente común de errores.

En nuestro ejemplo tenemos, por tanto, que definir un constructor copia que realice correctamente la copia del atributo *numero*.

```
Telefono::Telefono (const Telefono &tel) {
  numero=new char[strlen(tel.numero)+1];
  assert(numero); //finaliza el programa si error memoria
  strcpy(numero, tel.numero);
  importe=tel.importe;
  cout <<"Constructor copia"<<endl;
}
```

De esta manera, la salida de nuestro programa tiene el siguiente aspecto:

```
Constructor por defecto
Constructor parametrizado
Constructor copia
Número: 555
Número: 927123456
Número: 927123456
Destructor
Número: 555
Número: 927123456
Destructor
Destructor
```

Ahora, sí aparecen tres llamadas a constructores cuando se crean los tres objetos. Además, el valor del atributo *numero* de *telefono2* no se ve afectado por la eliminación del objeto *telefono3*.

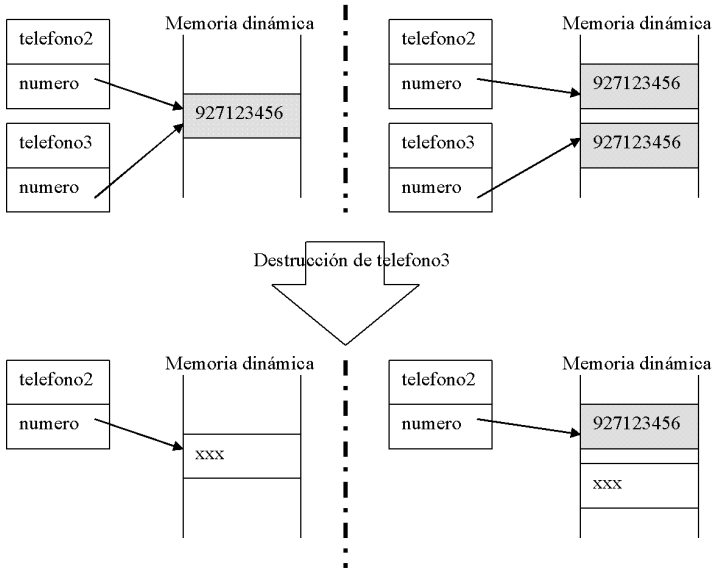


Figura 2.1: Diferencia entre copia bit a bit y constructor copia

2.3.4. Creación y eliminación dinámica de objetos

Al igual que para el resto de variables, existen dos formas de crear objetos: **estática y dinámicamente**. En el primer caso, los objetos se crean en ciertas partes del programa y son destruidos cuando salen de su ámbito. El compilador controla automáticamente su creación y destrucción. Este tipo de objetos son los que hemos visto hasta ahora. En el caso dinámico, es el propio programador el que controla su creación y destrucción mediante el uso de punteros a objetos.

A la hora de crear objetos dinámicamente debemos utilizar, como para el resto de variables dinámicas en C++, los operadores **new** y **delete**. Mediante **new** se crea un objeto dinámicamente y se invoca a su constructor, reservando la memoria necesaria para almacenar los atributos de dicho objeto. Con el operador **delete** se invoca al destructor del objeto y se libera dicha memoria. Por lo tanto, el programador puede controlar el ciclo de vida de un objeto explícitamente a través de estos operadores. Utilizando la clase *Telefono*, podremos definir objetos dinámicos de la siguiente forma:

```
Telefono *telefono3;
telefono3 = new Telefono;
Telefono *telefono3 = new Telefono ("927123456","Di","Juan");
```

La operación devuelve un puntero del tipo apropiado que se puede utilizar para acceder a los miembros del objeto correspondiente. En este ejemplo, el acceso

a los miembros del puntero a objeto *telefono3* se realiza de la misma forma que se accede desde un puntero a los campos de una estructura, utilizaremos el operador '->' en lugar del operador '.', usado en la definición de un objeto estático.

```
Telefono *telefono3=new Telefono ("927123456","Di","Juan");
telefono3->actualizarImporte(30);
cout<<"Importe: "<< telefono3->obtenerImporte()<<endl;
telefono3->mostrar();
```

Podemos además declarar un array de punteros a objetos, de forma que en cada posición del array existe un puntero a un objeto de la clase definida.

```
Telefono *telefono4[2];
telefono4[0]=new telefono("927654321", "No", "María Paz");
telefono4[0]->actualizarImporte(20);
cout<<"Importe: "<< telefono4[0]->obtenerImporte()<<endl;
telefono4[0]->mostrar();
telefono4[1]=new telefono("927555555", "No", "Jesús Sanz");
telefono4[1]->actualizarImporte(50);
cout<<"Importe: "<< telefono4[1]->obtenerImporte()<<endl;
telefono4[1]->mostrar();
```

En este ejemplo se ha declarado un array *telefono4* de dos posiciones en cada una de las cuales se almacenará un puntero a un objeto de tipo *Telefono*. De esta forma, debemos crear dinámicamente en cada posición un objeto y más tarde acceder a sus miembros.

Para liberar la memoria ocupada por un objeto creado dinámicamente utilizamos el operador delete. Al usar **delete** sobre un puntero a un objeto, el compilador invoca al destructor del objeto, el cual realiza las operaciones definidas en él antes de liberar la memoria ocupada por el puntero a dicho objeto.

En los ejemplos mostrados anteriormente, eliminaríamos los punteros a objetos creados de la siguiente forma:

```
delete telefono3;
delete [] telefono4;
```

2.4. Paso de objetos como parámetros a funciones

Los objetos se pueden pasar como parámetros a funciones de la misma forma que se pasa otro tipo cualquiera de variable. Existen dos formas de pasar argumentos a una función, el paso por referencia y el paso por valor.

El paso de parámetros por defecto en C++ es el paso **por valor**. Si pasamos una variable como parámetro por valor, el compilador genera una copia de la variable para que sea utilizada por la función, evitando que se modifique la variable original. El paso **por referencia** significa que no se pasa la variable como tal sino una referencia a la misma. Este tipo de paso de parámetros se suele utilizar por dos motivos básicos: uno, porque es más eficiente y, dos, porque se desea modificar la variable desde dentro de la función. A su vez, existen dos formas de pasar una variable como parámetro por referencia: como un puntero, es el formato heredado de C, y como referencia, formato introducido por C++.

Veamos un ejemplo:

```
void funcion(int uno, int * dos, int& tres);
```

Esta función declara tres parámetros distintos. El primero indica que el paso del parámetro *uno* se hace por valor. Por su parte, los parámetros *dos* y *tres* se pasan por referencia: el dos, al formato de C, y el tres, al formato de C++.

2.4.1. Paso de objetos por valor

La manera por defecto en C++ de pasar objetos como parámetros a una función es utilizando el paso por valor. En este caso, cuando se pasa un objeto a una función se crea una copia del mismo. El hecho de que se cree una copia del nuevo objeto puede plantear la duda de si se llama al constructor del nuevo objeto y al destructor del mismo. Veremos la respuesta a esta pregunta con un ejemplo:

```
class Entero {
  int valor;
public:
  Entero();
  Entero(int aux);
  void modificarValor(int _valor) {
    valor= _valor;
  }
  int obtener_Valor() {
    return valor;
  }
  ~Entero();
};
Entero::Entero() {
  valor=0;
  cout << "Construyendo entero con valor: "<<valor<<endl;
}
Entero::Entero(int _valor) {
```

```
  valor= _valor;
  cout << "Construyendo entero con valor: "<<valor<<endl;
}
Entero::~Entero() {
  cout << "Destruyendo entero con valor: "<<valor<<endl;
}
void cambiaValor(Entero objaux) {
  objaux.modificar_Valor(5);
  cout << "Dentro de cambiaValor: ";
  cout << objaux.obtener_Valor()<<endl;
}
void main() {
  Entero num1(1);
  cambiaValor(num1);
  cout << "En main: " << num1.obtener_Valor()<<endl;
}
```

El resultado de la ejecución de este ejemplo es el siguiente:

```
Construyendo entero con valor: 1
Dentro de cambiaValor: 5
Destruyendo entero con valor: 5
En main: 1
Destruyendo entero con valor: 1
```

Como se ve en el ejemplo, se llama una vez al constructor y dos veces al destructor. Realmente, el compilador invoca a dos constructores, pero, en el segundo caso, invoca al constructor copia. En este ejemplo no se ha implementado el constructor copia; por lo tanto, el compilador sintetiza uno que crea una copia del objeto a nivel de bit. Esta copia es la que la función *cambiaValor* modifica, permaneciendo el objeto *num1* inalterado.

La creación de una copia bit a bit de un objeto no es un efecto muy deseable. Supongamos que el objeto que se pasa por valor a la función reserva memoria dinámicamente y su destructor libera esta memoria automáticamente. Al crear una copia bit a bit del objeto, los atributos miembro punteros del nuevo objeto creado apuntan a la misma memoria que los del objeto pasado por valor. Debido a este hecho y a que al salir de la función se invoca al destructor de la copia del objeto, esta memoria será liberada y se perderán, por lo tanto, los valores del objeto inicial. Para evitar este tipo de situaciones puede utilizarse el constructor copia, que utiliza el compilador cuando un objeto se utiliza para inicializar otro. Como se ha comentado anteriormente, cuando existe un constructor de copia se evita la copia a nivel de bit.

En nuestro ejemplo el constructor copia quedaría así:

```
Entero::Entero(const Entero& entero) {
```

```
    valor= entero.valor;
    cout << "Construyendo copia entero valor: "<<valor<<endl;
}
```

La salida del programa sería ahora:

```
Construyendo entero con valor: 1
Construyendo copia entero valor: 1
Dentro de cambiaValor: 5
Destruyendo entero con valor: 5
En main: 1
Destruyendo entero con valor: 1
```

Al crear nosotros el constructor copia, se evita que el compilador haga una copia bit a bit.

El desconocimiento de este comportamiento del compilador de C++ por parte de los programadores inexpertos suele ser fuente de gran cantidad de errores difíciles de detectar. Por lo tanto, se recomienda el paso de objetos como parámetros por referencia para evitar tales situaciones de error. Existen técnicas para evitar que un objeto se pueda pasar como parámetro por valor a una función, pero su explicación está fuera del alcance de este libro.

2.4.2. Paso de objetos por referencia

El formato más utilizado de paso de objetos como parámetros de una función es el paso de parámetros por referencia. La razón básica responde a cuestiones de eficiencia. Al pasar 1 objeto por referencia, no es necesario que se realice una copia de todo el contenido del objeto. Por lo tanto, se gana en eficiencia. Consideremos, por ejemplo, el coste que conlleva pasar un objeto de la clase *Telefono* por valor. En este ejemplo, cada vez que invocásemos a la función a la que se le pasa como parámero por valor el objeto de la clase *Telefono*, deberíamos hacer una copia de todos los atributos miembro del objeto: *numero*, *usuario*, *importe* y *tipo_tarifa*. Si suponemos que esa función hipotética es llamada unas cien veces a lo largo de la aplicación, estaríamos realizando cien copias de objetos de la clase *Telefono*.

El único inconveniente del paso de parámetros por referencia es que la función que recibe este parámetro tiene la posibilidad de modificarlo. Sin embargo, no siempre vamos a desear que se permita esta posibilidad. Por esta razón, el lenguaje nos proporciona un modificador que se puede aplicar al parámetro por referencia para indicar que no se debe permitir su modificación dentro de la función. Este modificador es **const**. La utilización de **const** en la declaración del parámetro asegurará que este no podrá ser modificado dentro del código de la función. Veamos un ejemplo de su uso:

```
class Entero {
```

```
  int valor;
public:
  Entero();
  Entero(int aux);
  void modificarValor(int _valor) {
    valor= _valor;
  }

  int obtener_Valor() {
    return valor;
  }
  ~Entero();
};
Entero::Entero() {
  valor=0;
  cout << "Construyendo entero con valor: "<<valor<<endl;
}
Entero::Entero(int _valor) {
  valor= _valor;
  cout << "Construyendo entero con valor: "<<valor<<endl;
}
Entero::~Entero() {
  cout << "Destruyendo entero con valor: "<<valor<<endl;
}
void cambiaValor1(Entero& objaux) {
  objaux.modificar_Valor(5);
  cout << "Dentro de cambiaValor1: ";
  cout << objaux.obtener_Valor()<<endl;
}
void cambiaValor2(const Entero& objaux) {
  objaux.modificar_Valor(10); //Error: no permitido
  cout << "Dentro de cambiaValor2: ";
  cout << objaux.obtener_Valor()<<endl;
}
void main() {
  Entero num1(1);
  cambiaValor1(num1);
  cout << "En main: " << num1.obtener_Valor()<<endl;
  cambiaValor2(num1);
}
```

El resultado de la ejecución de este ejemplo es el siguiente:

```
Construyendo entero con valor: 1
```

```
Dentro de cambiaValor1: 5
En main: 5
Dentro de cambiaValor2: 5
Destruyendo entero con valor: 5
```

En este ejemplo, las funciones *cambiaValor1* y *cambiaValor2* reciben un objeto de la clase *Entero* como parámetro por referencia. De esta manera, como se puede ver en la salida del ejemplo, el compilador no intenta hacer una copia del objeto que se pasa por parámetro. Sólo se llama al constructor cuando se crea el objeto *num1* y al destructor cuando se alcanza el final del programa. En el caso de *cambiaValor1*, se permite modificar el valor del objeto pasado por parámetro; sin embargo, el compilador no permitirá modificación alguna en el caso de *cambiaValor2* porque se ha utilizado una referencia constante para el paso del objeto. De hecho el compilador dará un error al llegar a la línea en la que se intenta modificar el objeto pasado como referencia constante.

El uso del paso de objetos como parámetros por referencia constante ofrece una forma eficiente y menos problemática de realizar el paso de parámetros a funciones.

2.5. Funciones que devuelven objetos

Además de recibir objetos como parámetros, una función puede también devolver un objeto de la misma forma que devuelve otro tipo de variable. En este caso, cabe hacer la misma consideración que para el paso de parámetros. Una función puede devolver un objeto por valor o por referencia.

2.5.1. Por referencia

Si se devuelve un objeto como referencia es necesario tener el mismo cuidado que si se devuelve un puntero. Debemos asegurarnos de que la memoria a la que apunta la referencia o el puntero no se libera cuando termina la función.

```
class Entero {
  int valor;
public:
  Entero()  { valor=0 };
  Entero(int _valor);
  void modificarValor(int _valor) {
    valor= _valor;
  }
  int obtenerValor() {
    return valor;
  }
```

```
  Entero* suma(Entero* op2);
  Entero& suma(Entero& op2);
  Entero& incremento();
  ~Entero()  { };
};
Entero* Entero::suma(Entero* op2) {
  int x = op2->obtenerValor();
  op2->modificarValor(valor+x);
  return op2; //bien
}
Entero& Entero::suma(Entero& op2) {
  int x = op2.obtenerValor();
  op2.modificarValor(valor+x);
  return op2; //bien
}
Entero& Entero::incremento() {
  Entero *resultado;
  resultado->modificarValor(valor+1);
  return resultado; //mal
}
void main() {
  Entero num1, num2(5);
  Entero *r1 = num1.suma(&num2);
  Entero r2 = num1.suma(num2);
  Entero r3 = num1.incremento();
}
```

Al ejemplo de la clase *Entero* le hemos añadido tres funciones que devuelven objetos de la clase *Entero*. La primera de ellas recibe un puntero a *Entero* como parámetro y devuelve un puntero a *Entero*. Este caso es correcto porque el puntero que se devuelve (la memoria a la que apunta) ha sido creado fuera de la función *suma*. Ocurre lo mismo con la siguiente función *suma* en la que se usan referencias en vez de punteros: una forma más elegante, pero menos explícita. El ámbito de la referencia que se devuelve, *op2*, es exterior a la función *suma*; por lo tanto, no se libera el espacio de memoria al que apunta esa referencia cuando se termina la función suma. En el caso de la función incremento, sin embargo, estamos cometiendo el error de intentar devolver una referencia a un objeto local a la función. Al ser local, este objeto se destruirá al terminar la función incremento y estaremos devolviendo una referencia a una zona de memoria no definida.

2.5.2. Por valor

Por otro lado, cuando una función devuelve un objeto por valor, se produce la misma situación que en el paso de parámetros por valor. El compilador creará una copia del objeto que se intenta devolver. Para lograr esta copia, el compilador invoca al constructor copia del objeto que se intenta devolver. En el caso, de que no exista constructor copia el compilador sintetiza uno que realiza la copia bit a bit. Esta forma de actuar conlleva los problemas que ya se han comentado anteriormente en este capítulo. Por lo tanto, siempre que se quiera devolver un objeto por valor debemos asegurarnos de escribir el constructor copia para su clase.

El problema es que podemos estar intentando devolver objetos de una clase que no hemos escrito nosotros y de la cual no poseemos el código fuente de la definición, siendo imposible que evitemos los problemas derivados de la copia bit a bit en algunos objetos. Por consiguiente, se recomienda evitar devolver objetos por valor. Además, es menos eficiente que devolver objetos por referencia.

No se ha incluido un ejemplo en este caso porque el comportamiento de invocación a los destructores es similar al caso de pasar un objeto como parámetro por valor.

2.6. Miembros estáticos

2.6.1. Atributos miembro estáticos

En ocasiones puede ser útil que todos los objetos de una clase compartan un atributo, de manera que, por ejemplo, puedan comunicarse a través de él. En C, se usaban variables globales, pero el uso de este tipo de variables acarrea muchos problemas. C++, por su parte, permite obtener el mismo resultado pero aprovechando las características de encapsulación que ofrecen las clases. De este modo, podemos declarar atributos miembro que compartan la misma posición de almacenamiento para todos los objetos de una clase. Para lograrlo simplemente es necesario anteponer el modificador **static** a la declaración del atributo que queremos compartir.

Veamos un ejemplo de uso de un atributo miembro estático.

```
class Inicializa {
  static int valor;
public:
  void cambiar(int _valor) {
    valor = _valor;
  }

  void mostrar() {
    cout << "vale = " << valor <<endl;
```

```
  }
};
int Inicializa::valor=1;
void main() {
  Inicializa objeto1, objeto2;
  objeto1.mostrar ();
  objeto2.mostrar ();
  objeto1.cambiar(6);
  objeto1.mostrar();
  objeto2.mostrar();
}
```

En la clase del ejemplo hemos declarado el atributo *valor* como estático. Fijémonos en que es necesario definir también dicho atributo, no sólo declararlo como en el caso de un atributo no estático. Para definirlo debemos referirnos a él a través del nombre de la clase y del operador de resolución de alcance. Además, podemos aprovechar la definición para asignarle el valor que debe contener por defecto.

El resultado que aparece por pantalla es el siguiente:

```
vale = 1
vale = 1
vale = 6
vale = 6
```

Como sólo existe una copia del atributo *valor* para todos los objetos de la clase *Inicializa*, cuando invocamos a la función *cambiar* sobre *objeto1*, se modifica la única copia del atributo *valor* y todos los objetos muestran el mismo valor.

Un uso algo más especializado del modificador **static** consiste en la creación de constantes. Si junto al modificador **static** usamos también **const**, estaremos creando una constante.

```
class Constantes{
  static const ancho = 640;
  static const alto = 480;
};
```

2.6.2. Funciones miembro estáticas

Al igual que sucede con los atributos miembro, puede ser interesante, en un momento dado, poder invocar a una función sin tener que crear ningún objeto. Por otra parte, también es importante que esa función no esté aislada de la estructura de clases de nuestro sistema. Con este fin, el lenguaje C++ permite declarar funciones miembro estáticas. El programador debe indicar que

desea que una determinada función sea estática anteponiendo a su declaración el modificador **static**.

Las funciones miembro estáticas pueden ser invocadas a partir de un objeto como el resto de funciones miembro, aunque es más normal invocarlas directamente a través del operador de resolución de alcance.

```
class Entero {
  int valor;
public:
  static bool esNatural(int entero){
    if (valor>0) return true; //Error
    return (entero >0);
  }
};
int main(){
  bool res1 = Entero::esNatural(5);
  Entero entero;
  bool res2 = entero.esNatural(3);
}
```

En este ejemplo, hemos definido la función miembro estática *esNatural* en la clase *Entero*. Como muestra el ejemplo, se produce un error si se intenta acceder al atributo miembro *valor*. De hecho, las funciones miembro estáticas sólo pueden acceder a atributos miembro estáticos. Este comportamiento es coherente; ya que, si se puede invocar a esa función sin usar un objeto, entonces el atributo miembro no estático no existirá. Por lo tanto, el compilador debe evitar que se produzca esta situación. En el ejemplo, en la función **main** se muestran distintas formas de invocar a la función *esNatural*. La forma más común es usar el operador de resolución de alcance más el nombre de la clase, como se hace en la primera llamada del ejemplo.

Otra característica peculiar de las funciones miembro estáticas es que son las únicas funciones miembro que no reciben el puntero **this**. Es obvio que no deben recibirlo; ya que, si recordamos, el puntero **this** apuntaba al objeto que se encontraba en ejecución, sobre el que se invocaba la llamada de la función miembro. En este caso no hay objeto, por lo tanto, tampoco hay puntero.

Este tipo de funciones se usa para implementar distintas técnicas de programación como, por ejemplo, el patrón de diseño *Singleton*. Para más información sobre los patrones de diseño se recomienda al lector consultar el apéndice de bibliografía.

2.7. Sobrecarga

La sobrecarga no es un principio de la programación orientada a objetos, sino que más bien se trata de un recurso sintáctico del lenguaje. En este apar-

tado, no obstante, vamos a considerar la sobrecarga desde la óptica de la programación orientada a objetos. En C++, es posible sobrecargar tanto funciones como operadores.

2.7.1. Funciones sobrecargadas

El lenguaje C++ permite definir funciones que tengan el mismo nombre. Esto es posible gracias a que C++ identifica a las funciones por su signatura, que está formada por el nombre de la función y el número y tipo de los parámetros de la función. Por lo tanto, es posible tener funciones con el mismo nombre siempre que se diferencien en el número o tipo de los parámetros.

Como ejemplo podemos recuperar la clase *Entero* presentada anteriormente en el apartado de paso de objetos como parámetros y completarla un poco más.

```
class Entero {
  int valor;
public:
  Entero();
  Entero(int _valor);
  Entero& suma(Entero& op2);
  Entero& suma(int op2);
  int suma(Entero& op2, int op3);
  ~Entero();
};
int main(){
  int x = 10;
  Entero ent,ent2(5);
  Entero ent3 = ent.suma(ent2);
  Entero ent4 = ent.suma(6);
  int x = ent.suma(ent4,x);
}
```

En este ejemplo, aparece la función miembro *suma* sobrecargada. Se aportan diferentes versiones de esta función variando el número y tipo de los parámetros. El compilador puede distinguir en cada llamada a la función *suma* a qué cuerpo de función invocar según el tipo de los parámetros. Realmente, ya hemos utilizado la sobrecarga de funciones a lo largo de este capítulo para la definición de distintos constructores para una misma clase.

2.7.2. Operadores sobrecargados

Al igual que C++ permite la sobrecarga de funciones, es decir la definición de funciones que tienen el mismo nombre pero realizan operaciones distintas, permite también la sobrecarga en operadores, es decir, la utilización de los

mismos operadores pero aplicados con un nuevo significado. Un ejemplo de sobrecarga es el operador "*", ya que se puede utilizar como operador que multiplica dos números y como operador para trabajar con punteros.

Cuando un nuevo operador es sobrecargado, posee un nuevo significado para la clase en la que se define, pero sigue conservando todos los significados anteriores. La utilización de la sobrecarga de operadores tiene la ventaja de simplificar el uso de una clase y la escritura del código; sin embargo, por otro lado, puede llegar a complicar demasiado la definición de la clase. Debe tenerse en cuenta este inconveniente a la hora de utilizar sobrecarga de operadores; ya que, si su uso añade excesiva complejidad o confusión a los programas, es mejor no usarla.

Para sobrecargar un operador se utiliza la palabra clave **operator**. Es conveniente tener en cuenta que uno de los operandos debe ser de la clase que redefine el operador. También hay que ser conscientes del número de parámetros que aceptará un operador. Los operadores pueden ser unarios, sólo aceptan un parámetro, o binarios. En el siguiente ejemplo se muestra la sobrecarga de operadores.

En realidad, el tema de la sobrecarga de operadores es bastante más complejo que lo comentado en este apartado. Si el lector está interesado en profundizar en este tema, le conviene revisar la bibliografía. El objetivo de este apartado es introducir brevemente la capacidad de sobrecargar operadores dentro del lenguaje C++ con el fin único de que el lector conozca su existencia.

```
class punto3D {
  int x, y, z;
public:
  punto3D(){x=y=z=0;}
  punto3D(int a, int b, int c);
  punto3D(const punto3D &p3);
  punto3D operator+(const punto3D& op2);
  punto3D operator+(int op2);
  int operator==(const punto3D& op2);
  punto3D &operator=(const punto3D& op2);

  ~punto3D(){};
};
punto3D::punto3D (int a, int b, int c){
  x=a;
  y=b;
  z=c;
}
punto3D::punto3D (const punto3D &p3){
  x=p3.x;
  y=p3.y;
```

```
  z=p3.z;
}
int punto3D::operator==(const punto3D& op2){
  if ((x==op2.x)&&(y==op2.y)&&(z==op2.z))
    return 1;
  return 0;
}
punto3D punto3D::operator+(const punto3D &op2){
  int a=x+op2.x;
  int b=y+op2.y;
  int c=z+op2.z;
  return punto3D(a,b,c); //objeto temporal
}
punto3D punto3D::operator+(int op2){
  int a=x+op2;
  int b=y+op2;
  int c=z+op2;
  return punto3D(a,b,c); //objeto temporal
}
punto3D &punto3D::operator=(const punto3D &op2){
  x=op2.x;
  y=op2.y;
  z=op2.z;
  return (*this);
}
void main (void) {
  punto3D p1(10,20,30), p2(1,2,3), p3;
  if (p1==p2)
    cout<<"Iguales"<<endl;
  else
    cout<<"Distintos"<<endl;
  //sobrecarga operador +
  p3=p1+p2;
  cout<<p3;
  // Multiples sumas
  p3=p1+p2+p3;
  cout<<p3;
  // Multiples asignaciones
  p3=p1=p2;
  cout<<p3;
  // Ahora no hay error:
  p3=p3+10;
  cout<<p3;
  // Pero aqui hay error:
```

```
    //p3=10+p3;
}
```

En este ejemplo, hemos sobrecargado los operadores *asignación*, *suma* y *comparación* para la clase *punto3D*. Todos son operadores binarios, por lo tanto, reciben un solo parámetro de entrada: el segundo operando. El primer operando es el objeto que está en ejecución, sobre el que se invoca la versión sobrecargada del operador correspondiente. Este hecho lo podemos observar en la implementación de cada una de las versiones de los operadores en las que se utilizan los atributos miembro del objeto actual. Otra característica interesante es la posibilidad de concatenar las llamadas a los operadores, como se muestra en el ejemplo con la suma y la asignación. También es importante tener en cuenta el orden de los operandos cuando se quiere usar un operador sobrecargado. Si nos fijamos, la última operación de suma del ejemplo da error. La razón de este error consiste en que hemos sobrecargado el operador *suma* para la clase *punto3D*; por lo tanto, el primer operando debe ser un objeto de esta clase.

2.8. Resumen

En este capítulo hemos sentado las bases para entender el paradigma de programación orientada a objetos. Se han presentado los conceptos básicos de objeto y clase, así como el principio de la encapsulación que es uno de los pilares de la programación orientada a objetos.

La creación y eliminación de objetos constituye uno de los problemas más comunes para los programadores novatos. Por lo tanto, se ha intentado exponer de forma práctica cada una de las características asociadas a este punto dentro del lenguaje C++. Especial atención han tenido los diferentes tipos de constructores, la explicación de la creación automática de objetos por parte del compilador y el paso de objetos como parámetros a funciones o su retorno.

Para terminar se ha realizado una somera introducción al concepto de sobrecarga, al que se regresará en capítulos posteriores.

cc BY NC SA

BY NC SA

Capítulo 3

Mecanismos de reutilización

Una de las principales motivaciones del origen de la programación orientada a objetos consiste en definir nuevos conceptos y abstracciones en los lenguajes de programación que faciliten la reutilización de código y favorezcan la extensibilidad de los sistemas software. La posibilidad de reutilizar código es un objetivo principal en cualquier equipo de desarrollo software; ya que, permite la construcción de nuevas aplicaciones o la modificación de aplicaciones existentes con menor esfuerzo y coste.

En la programación estructurada no se introducen mecanismos explícitos de reutilización de código; por lo tanto, las técnicas utilizadas para la reutilización no iban mucho más allá del tradicional cortar y pegar. Evidentemente, ésta no es una técnica de reutilización demasiado fiable. Además, tampoco se puede usar para la reutilización de grandes porciones de código.

Para solventar en parte esta carencia, surge la POO. Como ya se ha comentado en capítulos anteriores, en la POO el código aparece estructurado en clases. La clase constituye el eje sobre el que se definen los mecanismos de reutilización de código. La reutilización de código consiste en crear nuevas clases a partir de clases ya construidas y depuradas, en vez de crearlas desde el principio: desarrollo incremental.

En la POO existen dos mecanismos de reutilización de código: la **composición** y la **herencia**. En lo referente a la composición, consiste simplemente en crear objetos de la clase reutilizada como atributos de la nueva clase. Se denomina composición porque la nueva clase está compuesta de objetos de clases que ya existían. En cuanto a la herencia, se basa en crear una nueva clase que especializa una clase existente, añadiéndole nuevo código sin modificar dicha clase base.

Una de las ventajas de la herencia y la composición es que posibilitan el desarrollo incremental permitiendo introducir nuevo código sin causar errores en el código existente. Si aparecen errores, éstos estarán aislados dentro del nuevo código, que es mucho más fácil y rápido de leer que si se hubiese modificado

el cuerpo del código existente.

Efectivamente, se produce una clara separación entre las clases. En C++, por ejemplo, no se necesita el código fuente de las funciones miembro para reutilizar su código, simplemente el fichero cabecera de la clase y el fichero objeto o librería con las funciones miembro compiladas.

La elección de la composición y la herencia como mecanismos de reutilización de código no se ha realizado de manera arbitraria, sino que se corresponde con la forma de clasificación que sigue la mente humana. La mente humana clasifica los conceptos de acuerdo a dos dimensiones: pertenencia y variedad. Desde este punto de vista, se puede decir que una cigüeña es un tipo de ave (variedad o una relación del tipo *es un*) y que una pluma es parte de un ave (pertenencia o una relación del tipo *tiene un*).

Es importante reseñar que el desarrollo de un programa es un proceso incremental, al igual que el aprendizaje humano. Podremos realizar todo el análisis deseado, y aun así no se conocerán todas las respuestas cuando se establezca el proyecto. Tendremos mucho más éxito si hacemos que nuestro proyecto vaya creciendo como una criatura orgánica y evolutiva, en lugar de construirlo de una vez como un rascacielos de cristal.

En resumen, a lo largo de este capítulo, aprenderemos a reutilizar clases ya creadas y depuradas en la construcción de clases más complejas y especializadas, siguiendo un proceso de desarrollo incremental y usando para ello los conceptos de composición y herencia introducidos por la POO.

3.1. Composición

Este apartado está organizado como sigue. En primer lugar, se propone un problema que se puede solucionar mediante composición. Posteriormente, se realiza una exposición de la composición a nivel conceptual. Y, finalmente, se introduce la composición a nivel de código en lenguaje C++, detallando los diferentes casos a tener en cuenta.

3.1.1. Problema

Imaginemos, por ejemplo, que queremos construir una clase que sea capaz de dibujar, trasladar y rotar en la pantalla un cubo en 3 dimensiones (cubo 3D). A simple vista puede parecer una tarea compleja. Y, posiblemente, lo sea si abordamos su implementación desde el principio sin ninguna base. Sin embargo, si pudiésemos reutilizar cierto código encargado de dibujar, trasladar y rotar un segmento en pantalla todo sería más fácil (desarrollo incremental). Realmente, un cubo 3D está formado por segmentos que intersectan entre sí: aristas. Por lo tanto, si ya poseemos la capacidad de dibujar segmentos, lo único que hay que hacer es reutilizar esa funcionalidad para dibujar segmentos que formen un cubo 3D. La complejidad de dibujar un cubo en 3D se reduce a la

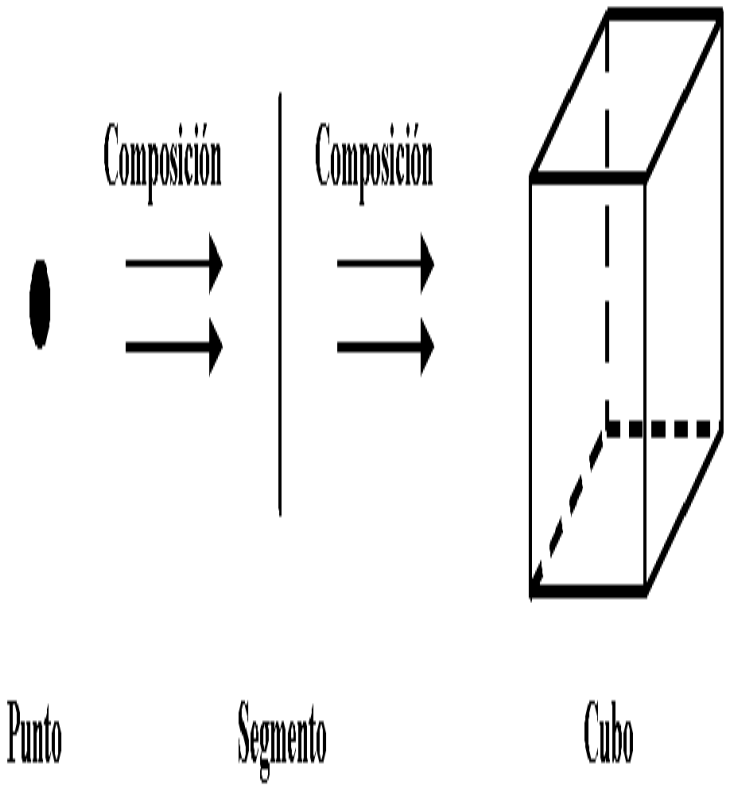


Figura 3.1: Mecanismo de composición

invocación de la funcionalidad de dibujo de cada una de las aristas que lo componen. No es necesario, entonces, volver a implementar complicadas rutinas de dibujo, sino que reutilizaremos las que ya teníamos perfectamente implementadas y probadas. Este mismo proceso de reutilización se puede seguir para la construcción de la clase segmento a partir de una clase existente previamente denominada punto 3D, como se aprecia en la figura 3.1.

Como se puede deducir fácilmente, una vez implementadas y depuradas las clases punto 3D y segmento 3D, pueden ser reutilizadas para la construcción de cualquier figura 3D. Al fin y al cabo, todas estas figuras están compuestas por segmentos y puntos 3D.

3.1.2. Concepto

La composición es el mecanismo más sencillo de reutilización de código de la POO. Consiste básicamente en crear nuevas clases a partir de clases existentes que actúan como elementos compositores de las nuevas. Dadas las características de modularidad de la POO, ni siquiera es necesario disponer del código fuente de las clases que vamos a usar en la composición de una nueva clase, es suficiente con poseer su declaración.

Finalmente, es necesario ser consciente de que el hecho de relacionar dos objetos mediante la composición implica que entre ellos existe una relación semántica *tiene un*. En el ejemplo anterior, existe esta relación entre segmento y punto 3D, un segmento tiene puntos 3D, y entre figura y segmento, una figura tiene segmentos. A la hora de decidir el uso de la composición como mecanismo de reutilización, es necesario asegurarse de que existe esta relación semántica entre los objetos implicados. En caso de que no se pueda establecer esta relación, no se debe usar la composición, sino que debe hacerse uso del mecanismo de programación adecuado a la situación concreta.

3.1.3. La composición en C++

El mecanismo de composición no posee ninguna sintaxis especial dentro de los lenguajes orientados a objeto (a partir de ahora LOO). No existen palabras reservadas dentro del lenguaje C++ para señalar la composición. La composición es un mecanismo de reutilización de código muy natural dentro de la estructura de código inherente a la POO. De hecho, se trata del mecanismo que hemos usado en todos los ejemplos que se han visto en el capítulo anterior, en los que hemos construido clases con atributos de tipos ya definidos (int, char, float...). De la misma manera, podemos construir clases a partir de tipos definidos por el usuario, o lo que es lo mismo, a partir de clases definidas por el usuario. Veamos, por ejemplo, cuál sería la implementación de la clase *Cubo3D* comentada anteriormente, partiendo de las clases *Segmento3D* y *Punto3D*. En el ejemplo sólo se muestran las interfaces de las clases *Segmento3D* y *Punto3D* porque se considera que estas clases ya han sido implementadas y vamos a reutilizarlas.

```
class Punto3D {

...
public:
  Punto3D(float x, float y, float z);
  void trasladar(int ix, int iy, int iz);
  void rotar(int gx, int gi, int gz);
...
};
class Segmento3D {
```

```
  Punto3D porigen, pfinal;

  ...
public:
  Segmento3D(const &Punto3D pori, const &Punto3D pfin);
  void trasladar(float ix, float iy, float iz);
  void rotar(int gx,int gy,int gz);
  void mostrar(void);
  void ocultar(void);
};
class Cubo3D {
  int numaristas;
  Segmento3D aristas[20];
public:
  Cubo3D(void);
  Cubo3D(float lado);
  void mostrar(void);
  void ocultar(void);
  void trasladar(float ix,float iy,float iz);
  void rotar(int gx,int gy,int gz);
  void introarista(const &Punto3D po, const &Punto3D pf);
};
void Cubo3D::trasladar(float ix,float iy,float iz) {
  int arista;
  for (arista=1; arista<=numaristas; arista++)
    aristas[arista].trasladar(ix,iy,iz);
}
...
void Cubo3D::Cubo3D(float lado) {
  float longi = lado/2;
  Punto3D A(-longi,longi,longi),  B(longi,longi,longi),
          C(longi,longi,-longi),  D(-longi,longi,-longi),
          E(-longi,-longi,longi), F(longi,-longi,longi),
          G(longi,-longi,-longi), H(-longi,-longi,-longi);
  introarista(H,G); introarista(H,E);
  introarista(E,F); introarista(G,F);
  introarista(A,B); introarista(D,A);
  introarista(C,D); introarista(C,B);
  introarista(G,C); introarista(D,H);
  introarista(F,B); introarista(A,E);
}
```

Como vemos en el ejemplo, la clase *Segmento3D* declara dos atributos miembro privados de la clase *Punto3D*. Mientras que la clase *Cubo3D* declara un vector

de *Segmento3D* como atributos miembro privados. Por lo tanto, estamos usando el mecanismo de composición para reutilizar el código de las clases *Segmento3D* y *Punto3D* en la construcción de la clase *Cubo3D*. Una de las grandes ventajas que obtenemos es la simplicidad en la implementación de las funciones de la clase *Cubo3D*. Por ejemplo, fijémonos en lo sencillo que resulta trasladar la posición del cubo, sencillamente sólo tenemos que trasladar cada una de las aristas que lo forman.

3.1.4. Interfaz de la clase contenedora

El hecho de que una clase (contenedora) esté formada a partir de otras (compositoras) no significa que esta clase deba declarar como públicas las clases de las que se compone. En realidad, si actuamos así, estamos yendo en contra del principio de encapsulación, uno de los pilares básicos de la POO. Por lo tanto, si deseamos seguir conservando la abstracción de datos, con todas sus ventajas, se deben declarar como privadas.

En algunas ocasiones, sin embargo, se necesita modificar la interfaz funcional de la clase contenedora, incluyendo algún método que trabaje con el objeto embebido. No obstante, esta modificación no debe consistir en una copia literal o parcial de la misma interfaz funcional de la clase compositora, sino que sólo deben introducirse métodos que ofrezcan una funcionalidad clara de la clase contenedora, no de la compositora. Esta forma de actuar toma especial relevancia cuando la clase compositora está formada a partir de clases que representan estructuras de datos.

Por ejemplo, imaginemos que disponemos de una clase *Colegio* encargada de gestionar la información de todos los profesores (clase *Profesor*) de un colegio. Para lograr este objetivo, la clase *Colegio* declara un atributo miembro *profesores* del tipo *ListaProfesores*, clase que implementa la estructura de datos lista. De esta forma, la clase *Colegio* hace uso de una estructura lista para mantener a los profesores que forman parte de un colegio determinado. Por otra parte, tenemos otra clase que representa a la administración de enseñanza, clase *Administracion*, que va a gestionar los diferentes colegios que dependen de ella, en nuestro ejemplo va a ser un único colegio, atributo *colegio*. La administración es la encargada de asignar profesorado a cada colegio y de requerir informes sobre el profesorado de cada colegio. Por lo tanto, necesita que la clase *Colegio* contemple en su interfaz esta funcionalidad referente a la gestión de los profesores. Vamos, a continuación, a ver el código de este ejemplo.

```
Class Profesor {
...
};
class ListaProfesores {
...
public:
```

```
  ListaProfesores();
  int insertar(const Profesor &elemento);
  int numElementos();
  Profesor &siguiente();
  int existe(const Profesor &elemento);
  int borrar(const Profesor &elemento);
};
class Colegio {
  ListaProfesores profesores;
public:
  Colegio();
  int altaProfesor(const Profesor &profesor) {
    if (!profesores.existe(profesor))
      return profesores.insertar(profesor);
    return 0;
  }
  int bajaProfesor(const Profesor &profesor) {
    return borrar(profesor);
  }
  void listarProfesores() {
    int cuantos, i;
    cuantos = profesores.numElementos();
      for (i=1; i< cuantos; i++)
        profesores.siguiente().mostrar();
  }
};
class Administracion {
  Colegio colegio;
public:
  void plantilla() {
    ...
    colegio.listarProfesores();
    ...
  }
};
```

Como vemos, la interfaz de la clase *Colegio* posee funciones miembro que interactúan con la lista de profesores, pero no exponen al exterior el tipo de estructura usada para el mantenimiento de los profesores. Estas funciones son *altaProfesor*, *bajaProfesor* y *listarProfesores*. De esta manera, se sigue manteniendo el principio de encapsulación y el código externo a la clase *Colegio* sigue siendo independiente del tipo de estructura que se usa dentro de esta clase para el mantenimiento de los profesores. Por lo tanto, se podría variar dentro de la clase *Colegio* el tipo de estructura usada para el mantenimiento de los

profesores sin tener que modificar el código externo a la misma: el código de la función *plantilla* de la clase *Administracion* no sufriría ningún cambio.

Veamos en nuestro ejemplo qué cambios se producen al usar una clase *ABO*, que implementa la estructura árbol binario ordenado, en vez de la clase *Lista-Profesores.*

```
class ABO {
   ...
public:
  ABO();
  ABO *hijoIzquierdo();
  ABO *hijoDerecho();
  Profesor &raiz();
  int insertar(const Profesor &elemento);
  int numElementos();
  int existe(const Profesor &elemento);
  int borrar(const Profesor &elemento);
};
class Colegio {
  ABO profesores;
  //Modificaciones internas a Colegio
  void inorden(ABO *profes);
public:
  Colegio();
  int altaProfesor(const Profesor &profesor);
  int bajaProfesor(const Profesor &profesor);
  //Modificaciones internas a Colegio
  void listarProfesores() {
     inorden(&profesores);
  }
};
class Administracion {
  Colegio colegio;
public:
  void plantilla() {
    ...
    colegio.listarProfesores();
    ...
  }
};
```

En nuestro ejemplo, el cambio del tipo de estructuras solamente produce cambios internos a la clase *Colegio*. Mientras que el resto del programa permanece inalterado; ya que, es independiente de los cambios que afecten internamente a la clase *Colegio*.

Hasta ahora hemos hablado de cómo se recomienda programar este tipo de situaciones. Veamos, ahora, un ejemplo de lo que no se debe hacer si queremos obtener un código que se beneficie de toda la potencia de la POO. En este caso, la clase *Colegio* introduce parcialmente en su interfaz la interfaz de la estructura que usa para mantener a los profesores.

Realmente, este caso es una solución parecida a declarar la estructura como un atributo público; sin embargo, suele ser un error común entre los principiantes de la POO suponer que están respetando el principio de encapsulación por el mero hecho de declarar como privados los atributos de la clase contenedora, mientras que están violando tal principio al modificar erróneamente la interfaz de la clase contenedora.

```
class Colegio {
  ListaProfesores profesores;
public:
  Colegio();
  //Interfaz de la clase Lista
  int insertar(const Profesor &elemento);
  int numElementos(); Profesor &siguiente();
  int existe(const Profesor &elemento);
  int borrar(const Profesor &elemento);
};
class Administracion {
  Colegio colegio;
public:
  void plantilla() {
    ...
    int cuantos, i;
    cuantos = colegio.numElementos();
    for (i=1; i< cuantos; i++)
      colegio.siguiente().mostrar();
    ...
  }
};
```

En este caso, la clase *Colegio* introduce en su interfaz las funciones *insertar*, *siguiente* y *numElementos*, que son funciones propias de la clase *ListaProfesores*. Esta manera de actuar provoca que el código de la función *plantilla* de la clase *Administracion* se haga dependiente de la estructura de datos usada en la clase *Colegio*. Por lo tanto, si realizamos el mismo cambio de estructura planteado anteriormente, se verá afectado el código de la función *plantilla*; ya que, la interfaz de la clase *Colegio* ha cambiado.

```
class Colegio {
  ABO profesores;
```

```
public:
  Colegio();
  //Interfaz de la clase ABO ABO *hijoIzquierdo();
  ABO *hijoDerecho();
  Profesor &raiz();
  int insertar(const Profesor &elemento);
  int numElementos();
  int existe(const Profesor &elemento);
  int borrar(const Profesor &elemento);
};
class Administracion {
  Colegio colegio;
  void inorden(Colegio *abo);
public:
  void plantilla() {
    ...
    inorden(&colegio);
    ...
  }
};
```

Como podemos apreciar en el código, en este caso, el cambio de estructura de datos en la clase *Colegio* supone realizar muchas más modificaciones que en el caso anterior. Ahora, los cambios no se limitan al interior de la clase *Colegio*, sino que afectan también a su interfaz. Al haberse modificado la interfaz de la clase *Colegio*, es necesario modificar todo el código que interactúe con objetos de esa clase para adaptarlo a la nueva interfaz. En definitiva, al no limitar los cambios al interior de la clase *Colegio*, se multiplican el número de modificaciones que hay que hacer en el código, aumentando también las posibilidades de introducir errores.

Para terminar este punto, es obligatorio hacer hincapié en que este modo de actuar no siempre se corresponderá con las necesidades de nuestros programas. En algunas situaciones, puede merecer la pena declarar como públicas algunas clases compositoras para, por ejemplo, disminuir la complejidad del diseño.

3.2. Herencia

La estructura de este apartado es similar a la del anterior. En primer lugar, se propone un problema que se puede solucionar mediante el mecanismo de la herencia. Posteriormente, se realiza una exposición de la herencia a nivel conceptual, explicando el contenido semántico de esta relación y la construcción de jerarquías. Y, finalmente, se introduce la herencia a nivel de código en lenguaje C++, detallando tanto la sintaxis como las diferentes consideraciones a tener en cuenta.

Antes de comenzar la explicación, es necesario hacer un breve comentario sobre la notación. Los términos clase base, superclase o clase padre hacen todos referencia a la clase de la que se hereda. Mientras que los términos clase derivada, subclase o clase hija hacen referencia a la clase que hereda.

3.2.1. Problema

Imaginemos, por ejemplo, que deseamos construir una clase *TelefonoMovil*. Y, en este caso, disponemos de otra clase, *Telefono*, que contiene la funcionalidad base. Sin embargo, cierta parte de la funcionalidad exigida para nuestra nueva clase no está presente en la clase original, por ejemplo el envío de mensajes cortos. Llegados a este punto, podríamos optar por tomar como base la clase original, *Telefono*, acceder a su código y extenderla directamente para adaptarla a los requisitos funcionales de la clase *TelefonoMovil*. Sin embargo, existen importantes razones que desaconsejan esta forma de actuar, tales como:

1. La clase base posiblemente haya seguido un riguroso proceso de prueba. Modificar la clase requeriría volver a realizar completamente dicho proceso de prueba para volver a comprobar el funcionamiento anterior y el añadido de la clase *TelefonoMovil*.

2. En el caso de que otros desarrolladores utilicen esta clase, es posible que no deseen disponer del comportamiento adicional (envío de mensajes cortos). La clase original puede ser todo lo que ellos necesitaban y modificarla sólo aumentaría la complejidad y el coste de la clase, reduciendo su eficiencia.

3. Puede que no sea recomendable o posible modificar el código de la clase original. Cualquier error introducido en esa modificación puede afectar seriamente a todos los programas que utilicen esa clase. Además, el código fuente de la definición de clase no siempre está disponible.

4. En general, modificar una clase, especialmente una que no ha escrito uno mismo, requiere un entendimiento completo de la implementación de la clase, y tal entendimiento puede ser imposible o poco práctico.

Los lenguajes de POO, proporcionan un mecanismo para la extensión del comportamiento de una clase. A este mecanismo se le llama **herencia**, y es considerado como uno de los pilares básicos de la programación orientada a objetos.

3.2.2. Concepto

Desde el punto de vista de la programación orientada a objetos, la herencia permite definir una nueva clase extendiendo una clase ya existente. Esta nueva clase, clase derivada, posee todas las características de la clase más general y, además, la extiende en algún sentido. Mediante la herencia, una clase derivada

hereda los atributos y métodos definidos por la clase base. De esta forma se evita volver a describir estos atributos y operaciones, obteniéndose así una importante reutilización de código.

Normalmente, las clases base suelen ser más generales que las clases derivadas. Esto es así porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian, concretan y particularizan. La clase derivada, por lo tanto, tiene la posibilidad de extender las características heredadas de la clase base. Y lo puede hacer mediante una de estas tres maneras básicas:

- añadiendo nuevos miembros: atributos y métodos,
- modificando la visibilidad de los atributos y métodos heredados,
- proporcionando una nueva implementación de las operaciones heredadas: redefinición.

De la misma forma que la composición conlleva una relación *tiene un* entre los objetos implicados, la herencia implica la existencia de una relación *es un* o *es un tipo de* entre la clase derivada y su clase base. Por lo tanto, a la hora de utilizar el mecanismo de la herencia hay que asegurarse de que se establece tal relación semántica entre la nueva clase y la clase original.

3.2.3. Jerarquías de clases

El mundo real está compuesto por un número inconcebible de clases de objetos como, por ejemplo, el coche, la bicicleta, la motocicleta, etc. Para reducir la complejidad que supone concebir tal cantidad de objetos, la mente humana utiliza un mecanismo básico de relación entre clases de objetos: la clasificación. Mediante la clasificación se pueden agrupar clases de objetos que comparten una esencia común. De esta manera se van creando grandes clasificaciones, por ejemplo, *Vehículos* sería una clase que agruparía a las anteriores, pero también podríamos introducir un mayor nivel de detalle al introducir dos nuevas clasificaciones: *Vehículos de dos ruedas* y *Vehículos de cuatro ruedas*.

El mundo de la programación puede entenderse en muchos casos como un reflejo del mundo real. En este caso, la POO imita el mecanismo de clasificación a través del concepto de herencia. De esta manera, relacionando clases mediante la herencia podemos construir jerarquías de clases, en las que una clase puede ser a la vez clase base y clase derivada. Es interesante tener en cuenta que según bajamos por la jerarquía vamos obteniendo clases cada vez más especializadas, mientras que según subimos son más genéricas. En el ejemplo anterior, podríamos construir una jerarquía de clases que imitase la clasificación de vehículos observada en el mundo real. Para la construcción de esta jerarquía caben dos aproximaciones básicas: generalización o especialización.

En el primer caso, partimos de una serie de objetos que contienen una raíz común, por ejemplo *Coche*, *Bicicleta*, *Avión* y *Cohete*. Extraemos esa raíz,

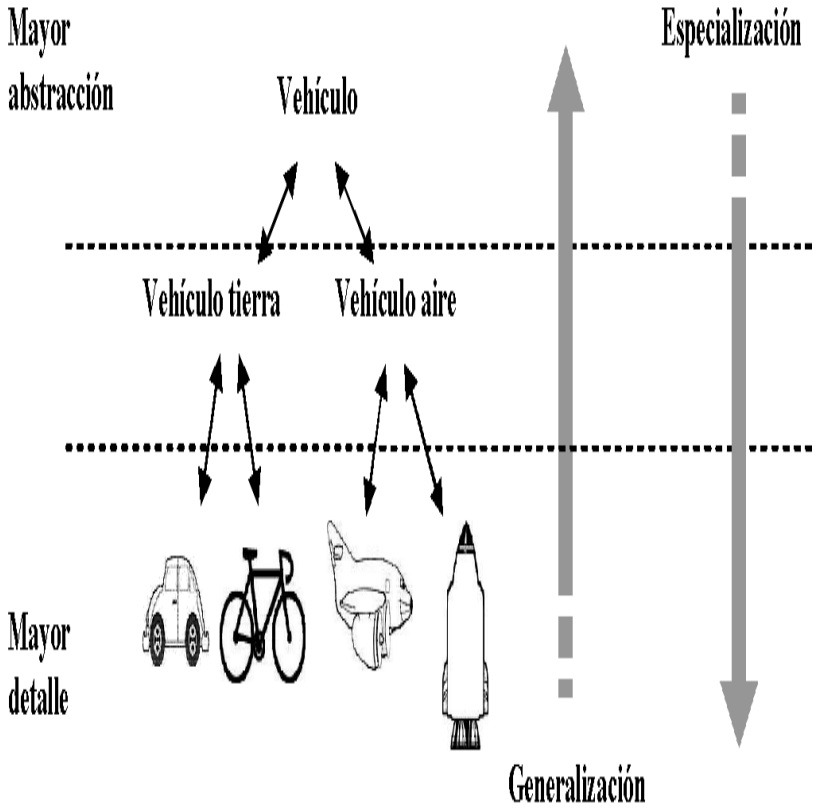


Figura 3.2: Jerarquía de vehículos

código común, para crear una clase base que la contenga y de la que hereden las clases de objetos iniciales. En este caso, detectamos dos raíces: *Vehículo de tierra*, clase base de *Coche* y *Bicicleta*, y *Vehículo de aire*, clase base de *Avión* y *Cohete*. De esta manera, ya tenemos dos pequeñas jerarquías, que son susceptibles de ser agrupadas en una sola; ya que, poseen una raíz común: *Vehículo*. Así, finalmente formamos la jerarquía que muestra la figura 3.2.

Esta misma jerarquía es posible obtenerla a través de la especialización. En este caso, el proceso es a la inversa. Partimos de una clase genérica, como es *Vehículo*, y vamos creando nuevas subclases que extienden sus características, *Vehículo de tierra* y *Vehículo de aire*. Evidentemente, podemos seguir extendiendo estas nuevas clases para obtener clases aun más especializadas, como

Coche o *Avión*.

En conclusión, las características comunes de las clases de la jerarquía pertenecerán a la clase base, por ejemplo una variable que indique su velocidad, la función de arrancar y la de frenar, etc. Mientras que las características particulares de alguna de ellas pertenecerán sólo a la subclase, por ejemplo el número de platos y piñones, que sólo tiene sentido para una bicicleta, o la función despegar que sólo se aplicaría a los *Vehículos de aire*.

Para hablar de herencia y jerarquías de clase estamos creando clases de objetos que aparecen de manera tangible en el mundo real; sin embargo, este reflejo tangible en la realidad no es un requisito indispensable, como ya sabemos, ni para la creación de clases, ni para el uso de la herencia, ni para la formación de jerarquías.

En definitiva, la herencia permite mediante la clasificación jerárquica gestionar de forma sencilla la abstracción y, por lo tanto, la complejidad.

3.2.4. La herencia en C++

Siguiendo la terminología de C++, la clase de la que se parte en la herencia recibe el nombre de clase base, y la nueva clase que se obtiene se denomina clase derivada.

3.2.4.1. Control de acceso a los miembros de la clase base

En particular vamos a reducir la explicación refiriéndonos sólo a los atributos miembro, pero la misma consideración es posible para las funciones miembro.

Como ya se vio en el capítulo 2, el lenguaje C++ proporciona una serie de palabras clave llamadas modificadores de acceso que sirven para especificar desde donde se puede acceder a un atributo miembro de una clase. Teníamos la posibilidad de hacer visibles o no los atributos de una clase desde el exterior mediante los modificadores **public** y **private**. El comportamiento más normal era declararlos como miembros privados respetando el principio de encapsulación de los datos.

Con la herencia surge una primera cuestión referente al acceso de los atributos miembro, ¿puede una función de una clase derivada acceder a los atributos miembro privados de su clase base? Si pensamos que es prioritario respetar el principio de encapsulación, la clase derivada no debería tener acceso directo a los atributos miembro de la clase base, sino a través de su interfaz funcional. Éste es el comportamiento que se produce cuando la clase base declara sus atributos como privados; ya que estos miembros no son heredados por la clase derivada. Veámoslo en un ejemplo.

```
class Documento{
  char autor[30];
```


CC BY NC SA

```
  char titulo[30];
public:
  void setAutor(char *_autor){
    strcpy(autor,_autor);
  }
  void setTitulo(char *_titulo){
    strcpy(titulo,_titulo);
  }
  char *getAutor(){
    return autor;
  }
  char *getTitulo(){
    return titulo;
  }
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  void setValores(char *_autor, char *_titulo,
                  char *_isbn, int _numpaginas) {
    strcpy(autor,_autor); //no permitido acceso directo
    strcpy(titulo,_titulo); //no permitido acceso directo
    setAutor(_autor); //acceso mediante funcion miembro
    setTitulo(_titulo); //acceso mediante funcion miembro
    strcpy(isbn,_isbn);
    numpaginas=_numpaginas;
  }
  char *getIsbn(){
    return isbn;
  }
  int getPaginas(){
    return numpaginas;
  }
};
```

En este ejemplo, aparecen dos clases: la clase base *Documento* y la clase derivada *Libro*. La clase *Documento* posee dos atributos miembro privados: *autor* y *titulo*. Al ser privados, la clase derivada no puede acceder a ellos directamente, como se puede comprobar en la función *setValores* de la clase *Libro*. De hecho, el compilador dará un error en estas líneas de código, avisándonos de que no es posible acceder directamente a esos atributos. Por lo tanto, el acceso tiene que realizarse a través de las funciones miembro de la clase base: *setAutor* y *setTitulo*.

En ocasiones, bien porque sea prioritario el rendimiento de la clase derivada o bien por cuestiones de efectividad, la clase derivada necesita acceder directamente a los atributos de la clase base. En este caso, podríamos optar por declarar como públicos los atributos de la clase base, permitiendo así el acceso desde la clase derivada. Esta opción, no obstante, no es la más recomendable; ya que, los atributos de la clase base son accesibles desde cualquier otra clase, no sólo desde la derivada, rompiendo totalmente el principio de encapsulación, como muestra el siguiente ejemplo.

```
class Documento{
public:
  char autor[30];
  char titulo[30];
  void setAutor(char *_autor){
    strcpy(autor,_autor);
  }

  void setTitulo(char *_titulo){
    strcpy(titulo,_titulo);
  }

  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  void setValores(char *_autor, char *_titulo,
                  char *_isbn, int _numpaginas) {
    strcpy(autor,_autor); //permitido acceso directo
    strcpy(titulo,_titulo); //permitido acceso directo
    strcpy(isbn,_isbn);
    numpaginas=_numpaginas;
  }

  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
};
class Biblioteca{
public:
  void archivarLibro(Libro &lib) {
    cout<<"Archivando libro "<<endl;
    //acceso directo: viola encapsulación
```

```
    cout<<"Titulo: "<<lib.titulo<<endl;
    //acceso directo: viola encapsulación
    cout<<"Autor: "<<lib.autor<<endl;
    //error no permitido acceso directo
    cout<<"ISBN: "<<lib.isbn<<endl;
    //error no permitido acceso directo
    cout<<"Pags: "<<lib.numpaginas<<endl;
    //acceso funcion miembro
    cout<<"ISBN: "<<lib.geIsbn()<<endl;
    cout<<"Pags: "<<lib.getPaginas()<<endl;
  }
};
```

En este ejemplo hemos añadido una clase adicional: la clase *Biblioteca*. Esta clase no deriva de ninguna de las otras dos. Por otra parte, hemos declarado los atributos de la clase *Documento* como públicos. De esta manera, se soluciona el problema anterior, se puede acceder directamente a ellos desde la función *setValores* de la clase derivada *Libro*. No obstante, se viola el principio de encapsulación; ya que, también es posible realizar ese acceso directo desde cualquier otra clase no relacionada mediante herencia, como es el caso de la función *archivarLibro* de la clase *Biblioteca*. En esta función, dada esta solución, se produce un efecto curioso: dado un objeto de la clase *Libro* es posible acceder directamente a los atributos que ha heredado, *autor* y *titulo*, pero no se permite este acceso a los atributos que declara él mismo, *isbn* y *numpaginas*. No parece, por tanto, que ésta sea una solución demasiado correcta.

Evidentemente, si no se puede solucionar mediante los modificadores **public** o **private**, el lenguaje C++ debe proporcionar alguna solución para este problema. Esta solución es el modificador **protected**. Si necesitamos que la clase derivada acceda a alguno de los atributos de la clase base, sólo tenemos que declararlos como protegidos. De esta manera serán accesibles desde las clases derivadas de la clase base, pero no desde el resto de las clases. Veamos cómo queda entonces nuestro ejemplo.

```
class Documento{
protected:
  char autor[30];
  char titulo[30];
public:
  void setAutor(char *_autor){
    strcpy(autor,_autor);
  }
  void setTitulo(char *_titulo){
    strcpy(titulo,_titulo);
  }
```

```
  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  void setValores(char *_autor, char *_titulo,
                  char *_isbn, int _numpaginas) {
    strcpy(autor,_autor); //permitido acceso directo
    strcpy(titulo,_titulo); //permitido acceso directo
    strcpy(isbn,_isbn);
    numpaginas=_numpaginas;
  }
  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
};
class Biblioteca{
public:
  void archivarLibro(Libro &lib) {
    cout<<"Archivando libro "<<endl;
    //error no permitido acceso directo
    cout<<"Titulo: "<<lib.titulo<<endl;
    cout<<"Autor: "<<lib.autor<<endl;
    cout<<"ISBN: "<<lib.isbn<<endl;
    cout<<"Pags: "<<lib.numPaginas<<endl;

    //acceso mediante funciones miembro
    cout<<"Titulo:"<<lib.getTitulo()<<endl;
    cout<<"Autor: "<<lib.getAutor()<<endl;
    cout<<"ISBN: "<<lib.geIsbn()<<endl;
    cout<<"Pags: "<<lib.getPaginas()<<endl;
  }
};
```

En este caso, los atributos de la clase base se han declarado como protegidos. Por lo tanto, el acceso directo a los mismos es posible desde la clase *Libro*, pero no, desde la clase *Biblioteca*. Ahora, el código del ejemplo es más coherente y no se producen efectos extraños como el comentado en el ejemplo anterior.

Como recomendación final, apuntar que es aconsejable heredar sólo la interfaz y el comportamiento de una clase, en vez de heredar el estado, es decir, sus atributos miembros. De esta manera se obtiene un diseño de clases menos dependiente y más extensible.

Clase base	C derivada public	C derivada protected		C derivada private
		C derivada	C no derivada	
private	No accesible	No accesible	No accesible	No accesible
protected	protected	protected	private	private
public	public	public	private	private

Cuadro 3.1: Comportamiento de los modificadores de acceso

3.2.4.2. Mayor control de acceso a los miembros de la clase base

En C++, es posible establecer un mayor control sobre la forma en que se heredan los miembros de la clase base. Este control se establecerá a la hora de definir el modo en que un clase derivada heredará de su clase base. El proceso de herencia puede llevarse a cabo de tres modos diferentes:

- primero, que la clase base sea **public** para la clase derivada
- segundo, que la clase base sea **protected** para la clase derivada
- tercero, que la clase base sea **private** para la clase derivada, es la opción por defecto en C++

En el primer caso, la clase derivada hereda los miembros **public** y **protected** de la clase base sin cambiar sus modificadores de acceso. En el segundo caso, los hereda como **public** y **protected** para el resto de clases que deriven de ella y como **private** para otras clases. Mientras que, en el tercer caso, la clase derivada los heredaría como **private**.

El comportamiento más habitual es utilizar la herencia **public**. La herencia **private** y **protected** aparecen en el lenguaje por razones de completitud más que por razones de utilidad.

El cuadro 3.1 muestra de forma resumida todas las posibilidades de acceso resultantes de mezclar las distintas posibilidades comentadas anteriormente.

3.2.4.3. Constructores y destructores en la herencia

Algunos elementos de la clase base no pueden ser heredados, los más importantes son los constructores y los destructores.

Haciendo un poco de memoria, los constructores son las funciones miembro de una clase que se invocan cada vez que se crea un nuevo objeto de esa clase. Su función básica consiste en inicializar los atributos miembro con los valores correspondientes. En cuanto a los destructores, serían la otra cara de la moneda. Son invocados en la destrucción de cada objeto y sirven para liberar los recursos que estaba utilizando un objeto determinado.

Cuando creamos un objeto de una clase que no deriva de ninguna otra, como ya se explicó, se invoca uno de los constructores de la clase del objeto. Ahora, cuando creamos un objeto de una clase que deriva de otra, ¿se invoca

sencillamente a uno de sus constructores o sucede algo más? Pensemos en el caso que presenta el ejemplo siguiente.

```
class Documento{
  char autor[30];
  char titulo[30];
public:
  Documento() {
    strcpy(autor,"anonimo");
    strcpy(titulo,"ninguno");
    cout<< "Creación documento por defecto"<<endl;
  }
  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
  ~Documento(){
    cout<<"Destrucción de documento"<<endl;
  }
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  Libro() {
    strcpy(isbn,"0000000000");
    numpaginas=0;
    cout<< "Creación libro por defecto"<<endl;
  }
  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
  ~Libro(){
    cout<<"Destrucción de libro"<<endl;
  }
};
int main() {
  Libro lib;//Llamada automatica a constructores
}
```

Tenemos dos clases: *Documento* y *Libro*. La clase *Documento* posee atributos miembro privados; con lo que, tales atributos no pueden ser inicializados desde el constructor de la clase derivada. Por lo tanto, el constructor de la clase base debe ser invocado primero para inicializar dichos atributos. Esto nos obliga a introducir una llamada al constructor de la clase base en el de la clase derivada. No obstante, si existe constructor por defecto en la clase base, el propio compilador de C++ se encarga de invocarlo automáticamente, en primer lugar. Podemos comprobar todo esto en la salida del programa:

```
Creación documento por defecto
Creación libro por defecto
Destrucción de libro
Destrucción de documento
```

En el caso de constructores parametrizados, se produce el mismo orden de invocación de los constructores. Sin embargo, ahora sí es necesario indicar explícitamente en el código la llamada a través del inicializador base. Este inicializador permite inicializar los atributos privados con los valores introducidos mediante el constructor parametrizado de la clase derivada. Veamos qué sucede si introducimos constructores parametrizados en las clases de nuestro ejemplo.

```
class Documento{
  char autor[30];
  char titulo[30];
public:
  Documento() {
    strcpy(autor,"anonimo");
    strcpy(titulo,"ninguno");
    cout<< "Creación documento por defecto"<<endl;
  }
  Documento(char *_autor, char *_titulo) {
    strcpy(autor,_autor);
    strcpy(titulo,_titulo);
    cout<<"Creación documento con parámetros"<<endl;
  }
  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
  ~Documento(){
    cout<<"Destrucción de documento"<<endl;
  }
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  Libro() {
    strcpy(isbn,"0000000000");
    numpaginas=0;
    cout<< "Creación libro por defecto"<<endl;
  }

  //constructor con inicializador base
  Libro(char *_autor, char *_titulo, char *isbn,
```

```
        int numpaginas) : Documento(_autor,_titulo) {
    strcpy(this->isbn,isbn);
    this->numpaginas=numpaginas;
    cout<<"Creación libro con parámetros"<<endl;
  }

  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
  ~Libro(){
    cout<<"Destrucción de libro"<<endl;
  }
};
int main() {
  //Llamada a constructores parametrizados
  Libro lib("Yo","POO","11111",545);
}
```

Como podemos observar, se ha introducido un constructor parametrizado en cada una de las clases: *Documento* y *Libro*. El inicializador base se especifica poniendo el carácter ":" detrás del constructor parametrizado de la clase *Libro* y después la invocación del constructor parametrizado de la clase *Documento*, pasando los argumentos recibidos como parámetros. La salida de este ejemplo será:

```
Creación documento con parámetros
Creación libro con parámetros
Destrucción de libro
Destrucción de documento
```

En definitiva, cuando creamos un objeto de una clase que deriva de otra, debemos ser conscientes de que se va a invocar primero el constructor de la clase base y luego el de la clase del objeto que estamos creando. Es muy importante darse cuenta de que dentro de una jerarquía de clases este comportamiento supone que se realicen llamadas a los constructores de todas las clases desde la clase raíz de la jerarquía hasta la clase de la que se quiere crear un objeto.

Por otra parte, algo similar acontece con los destructores. Cuando se destruye un objeto de una clase derivada, además de invocar a su destructor también se invoca al destructor de la clase base. En este caso, no obstante, el orden de las llamadas es el contrario, es decir, primero se invoca el de la clase derivada y luego el de la clase base. La razón de este orden consiste en que el destructor de la clase base puede intentar liberar recursos de los que dependen otros usados en la clase derivada. Veamos todo el comportamiento completo creando un nivel más en la jeraquía del ejemplo.

```
class Documento{
```

```
  char autor[30];
  char titulo[30];
public:
  Documento() {
    strcpy(autor,"anonimo");
    strcpy(titulo,"ninguno");
    cout<< "Creación documento por defecto"<<endl;
  }
  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
  ~Documento(){
    cout<<"Destrucción de documento"<<endl;
  }
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  Libro() {
   strcpy(isbn,"0000000000");
   numpaginas=0;
   cout<< "Creación libro por defecto"<<endl;
  }
  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
  ~Libro(){
    cout<<"Destrucción de libro"<<endl;
  }
};
class LibroInfantil: public Libro{
  int edad;
public:
  LibroInfantil() {
    edad = 3;
    cout<< "Creación libro Infantil por defecto"<<endl;
  }
  int calculoPalabras(){ return numpaginas * 100; }
  ~LibroInfantil(){
    cout<<"Destrucción de libro infantil"<<endl;
  }
};
int main() {
  //Llamada a constructores
  LibroInfantil *lib = new LibroInfantil();
```

```
  delete lib; //Llamada a destructores
}
```

En este caso, hemos introducido la clase *LibroInfantil* que deriva de la clase *Libro*. Y en la función main estamos creando y destruyendo un objeto de la clase *LibroInfantil*. A continuación se muestra la salida del ejemplo, en la que se aprecia el orden de invocación de los constructores y de los destructores.

```
Creación documento por defecto
Creación libro por defecto
Creación libro Infantil por defecto
Destrucción de libro infantil
Destrucción de libro
Destrucción de documento
```

3.2.4.4. Redefinición

La clase derivada, como hemos visto, va extender de alguna forma el comportamiento heredado de la clase base. Para ello, lo más común es la introducción de nuevas funciones miembro que complementan a las heredadas. Por ejemplo, en el ejemplo del apartado anterior, la subclase *LibroInfantil* introducía una nueva funcionalidad que no poseía su clase base: *calculoPalabras*.

Por otra parte, también es posible que la clase derivada necesite modificar o extender el comportamiento de una de las funcionalidades heredadas de la clase base. La clase derivada proporciona una nueva definición para una función miembro heredada. Este mecanismo recibe el nombre de redefinición.

```
class Documento{
  char autor[30];
  char titulo[30];
public:
  Documento() {
    strcpy(autor,"anonimo");
    strcpy(titulo,"ninguno");
    cout<< "Creación documento por defecto"<<endl;
  }
  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
  void mostrar() {
    cout<<"Mostrando documento"<<endl;
    cout<<"Titulo: "<<titulo<<endl;
    cout<<"Autor: "<<autor<<endl;
  }
  ~Documento(){
    cout<<"Destrucción de documento"<<endl;
```

```
  }
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  Libro() {
    strcpy(isbn,"0000000000");
    numpaginas=0;
    cout<< "Creación libro por defecto"<<endl;
  }
  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
  void mostrar() {
    cout<<"Mostrando libro"<<endl;
    cout<<"Titulo: "<<getTitulo()<<endl;
    cout<<"Autor: "<<getAutor()<<endl;
    cout<<"ISBN: "<<isbn<<endl;
    cout<<"Pags: "<<numpaginas<<endl;
  }
  ~Libro(){
    cout<<"Destrucción de libro"<<endl;
  }
};
int main() {
  Documento doc;
  Libro lib;
  doc.mostrar();
  lib.mostrar();
}
```

En este caso, la clase *Libro* redefine la función heredada *mostrar*. Como muestra la salida del ejemplo, cuando se crea una instancia de la clase base y se invoca la función miembro *mostrar*, se obtiene la salida de la función *mostrar* de *Documento*. Mientras que cuando se crea una instancia de la subclase y se invoca la función *mostrar* se invoca el cuerpo definido en *Libro*. Por lo tanto, en el ejemplo, la clase *Libro* redefine la función miembro heredada *mostrar*. La salida de este ejemplo será:

```
Creación de documento por defecto
Creación de libro por defecto
Mostrando documento
Titulo: ninguno
Autor: anonimo
```

```
Mostrando libro
Titulo: ninguno
Autor: anonimo
ISBN: 0000000000
Pags: 0
Destrucción de libro
Destrucción de documento
```

3.2.4.5. Conversiones entre objetos de la clase base y objetos de las clases derivadas

Cuando se crea una clase a partir de otra, se establece entre ellas una relación que se puede resumir en: la nueva clase es un tipo de la clase inicial. Dado que la herencia significa que todas las funciones de la clase base están disponibles en la clase derivada, cualquier mensaje que se pueda mandar a la clase base se puede mandar también a la clase derivada. Ver ejemplo:

```
class Documento{
  char autor[30];
  char titulo[30];
public:
  Documento() {
    strcpy(autor,"anonimo");
    strcpy(titulo,"ninguno");
    cout<< "Creación documento por defecto"<<endl;
  }
  int getClaveArchivo() { return 323; }
  char *getAutor(){ return autor;}
  char *getTitulo(){ return titulo;}
  ~Documento(){
    cout<<"Destrucción de documento"<<endl;
  }
};
class Libro: public Documento{
  char isbn[12];
  int numpaginas;
public:
  Libro() {
    strcpy(isbn,"0000000000");
    numpaginas=0;
    cout<< "Creación libro por defecto"<<endl;
  }
  char *getIsbn(){ return isbn;}
  int getPaginas(){ return numpaginas;}
```

```
  ~Libro(){
    cout<<"Destrucción de libro"<<endl;
  }
};
class Biblioteca{
public:
  void archivar(Documento &doc) {
    ...
    codigo = doc.getClaveArchivo();
    ...
  }
};
void main() {
  Biblioteca biblio;
  Libro lib;
  biblio.archivar(lib); //upcasting
}
```

Fijémonos en la función *archivar* de la clase *Biblioteca*, que acepta por parámetro una referencia de la clase *Documento*. Sin embargo, en la función main estamos invocando a la función *archivar* pasándole como parámetro una referencia a un objeto de la clase *Libro*. Aunque, a primera vista, pueda parecer un comportamiento extraño de un lenguaje tipado como es C++, todo se aclara si consideramos que un objeto de la clase *Libro* es a su vez un objeto de la clase *Documento*. Cualquier método que la función *archivar* pueda invocar sobre un ojeto de la clase *Documento*, lo puede hacer también sobre un objeto de la clase *Libro*; por lo tanto, el código de la función *archivar* es válido tanto para un objeto de la clase *Documento* como para los objetos de sus clases derivadas. La conversión de un objeto de una clase derivada a un objeto de una clase base se denomina **upcasting**.

El upcasting se realiza de lo más especializado, clase derivada, a lo más genérico, clase base; pero no, al revés. Un objeto de una clase base no se puede convertir en un objeto de una clase derivada; ya que, pueden existir atributos o funciones en la clase derivada que no estén presentes en la clase base.

3.2.4.6. Herencia múltiple

El lenguaje C++ permite la herencia múltiple. Una clase puede derivar de varias clases base. La sintaxis es la misma que la utilizada para la herencia simple, separando por comas cada una de las clases base de las que hereda.

Aunque el lenguaje nos brinde la posibilidad de utilizar la herencia múltiple, existe un fuerte debate en lo referente a si tiene sentido o no usarla dentro de un diseño software. Las principales razones que desaconsejan su uso es que introduce mayor oscuridad en el código sin aportar ningún beneficio que no

se pueda lograr usando sólo herencia simple. Ésta parece ser la postura más aceptada hoy día y así lo reflejan los nuevos lenguajes, como JAVA, en los que ha desaparecido la herencia múltiple.

3.2.4.7. Sobrecarga y redefinición

En lo referente a la sobrecarga, ya vimos en el capítulo anterior qué significaba y cómo se lograba, apuntando también que no se trata de un concepto de orientación a objetos, sino más bien de una característica del lenguaje C++. Resumiendo, esta característica nos permitía crear varias funciones con el mismo nombre, pero con diferentes listas de parámetros o tipos de retorno. Por su parte, la redefinición, concepto introducido en este capítulo, nos permite crear clases derivadas que extiendan el comportamiento de la clase base, cambiando la definición de las funciones heredadas. Por lo tanto, hemos visto cómo se comportan estos dos mecanismos por separado; sin embargo, nos falta saber cómo se comportarán cuando aparezcan juntos en el código de un programa. Para explicar su comportamiento vamos a recurrir, una vez más, a nuestro ejemplo de clasificación de documentos.

```
class Documento{
public:
  int archivar(){
    cout<<"Documento archivado"<<endl;
    return 1;
  }
  int archivar(int estanteria){
    cout<<"Documento archivado en "<<estanteria<<endl;
    return 1;
  }
};
class Libro: public Documento{
public:
  //Redefinición de una función sobrecargada
  int archivar(){
    cout<<"Libro archivado"<<endl;
    return 1;
  }
};
class Articulo: public Documento{
public:
  //Nueva versión: cambio de parámetros
  int archivar(char *clave){
    cout<<"Articulo archivado por "<<clave<<endl;
    return 1;
```

```
  }
};
class Informe: public Documento{
public:
  //Nueva versión: cambio tipo de retorno
  void archivar(){
    cout<<"Informe archivado" <<endl;
  }
};
```

En este caso tenemos la clase base, *Documento*, y tres clases que derivan de ella: *Libro*, *Articulo* e *Informe*. La clase *Documento* define dos versiones de la función *archivar*, es decir, esta función aparece sobrecargada. Por su parte, la clase *Libro* redefine una de las versiones de la función *archivar*. Mientras que las otras dos clases proporcionan una nueva versión de esa función: la primera, variando el número de parámetros y, la segunda, el tipo de retorno de la función. Ahora, la cuestión consiste en saber qué versiones de la función *archivar* están disponibles para los objetos de cada una de las tres clases derivadas.

```
int main() {
  int x;
  Libro libro;
  x = libro.archivar();  //error: versión no disponible
  x = libro.archivar(6);
  Articulo articulo;
  x = articulo.archivar();  //error: versión no disponible
  x = articulo.archivar("A");
  Informe informe;
  x = informe.archivar();  //error: versión no disponible
  informe.archivar();
};
```

Como se puede comprobar en el ejemplo, debemos ser conscientes de que la mezcla de sobrecarga y redefinición conlleva un comportamiento característico. En el caso del objeto de la clase *Libro*, la redefinición de una de las versiones de la función *archivar* supone que la otra versión pase a ser inaccesible. Por otro lado, como muestran las clases *Articulo* e *Informe*, si la clase derivada define una nueva versión de la función *archivar*, las versiones definidas en la clase base se hacen inaccesibles.

3.3. Resumen

En este capítulo hemos visto los mecanismos que provee la programación orientada a objetos para la reutilización de código: la composición y la herencia.

Básicamente, ambos mecanismos nos permiten crear nuevas clases o tipos a partir de clases ya definidas. La composición conlleva una relación *es parte de* entre los objetos relacionados. Mientras que la herencia supone una relación *es un tipo de*.

La elección entre composición y herencia no es un problema trivial en muchos de los programas desarrollados en base a objetos. El desarrollador se ve obligado a replantearse continuamente las jerarquías de clases y las composiciones creadas con el objetivo de asegurar una colección de clases óptima para el problema a resolver.

Capítulo 4

Polimorfismo

4.1. Concepto

El polimorfismo es el tercer pilar de la programación orientada a objetos. Supone un paso más en la separación entre la interfaz y la implementación. Permite mejorar la organización del código y simplificar la programación. Además, aumenta las posibilidades de extensión y evolución de los programas.

Está directamente relacionado con las jerarquías de clase. Básicamente, nos va a permitir que unos objetos tomen el comportamiento de objetos que se encuentran más abajo en la jerarquía, aumentando enormemente la expresividad del lenguaje. En definitiva, se trata de la posibilidad de que la identificación del tipo de un objeto se haga en tiempo de ejecución en vez de en tiempo de compilación. De esta manera, se pueden, incluso, construir estructuras en las que cada uno de sus elementos es de un tipo diferente.

Figura 4.1: Array de Instrumentos

Consideremos, por ejemplo, que hemos creado una jerarquía de clases de instrumentos musicales. Todos los instrumentos tienen la función *tocar*, pero cada uno tocará de una manera diferente. Mediante el polimorfismo vamos a conseguir crear, por ejemplo, un array de instrumentos como el que se muestra en la figura 4.1. Además, como veremos más adelante, podemos tratar a los diferentes instrumentos de la misma manera, sin tener que hacer distinción entre sus diferentes tipos; sin embargo, vamos a obtener un comportamiento diferente para cada uno de ellos. Si creamos, por ejemplo, un bucle que recorra este array de instrumentos invocando a la función *tocar*, esta función se comportará de

manera distinta para cada uno: no es lo mismo tocar una violín que tocar un piano.

4.2. Polimorfismo en C++

El polimorfismo en C++ se logra utilizando direcciones a objeto, tanto punteros como referencias. Básicamente, permite que una dirección a un objeto base se comporte como si fuese una dirección a un objeto derivado dentro de la misma jerarquía.

Antes de entrar en materia, el siguiente ejemplo pretende hacer una demostración práctica de la potencia que aporta el polimorfismo al lenguaje de programación. Empecemos por lo conocido, el ejemplo muestra dos clases: clase *PuestoTrabajo*, clase base, y la clase *Profesor*, clase derivada. Ambas clases están relacionadas mediante el mecanismo de la herencia, concepto estudiado en el capítulo anterior. Por otra parte, tenemos la función *diaLaborable* que recibe por parámetro una referencia del objeto base e invoca a la función *trabajar* sobre el argumento recibido. Esta función está definida en la clase base y redefinida en la clase derivada. Nuestra idea inicial, al escribir este código, era que la función *diaLaborable* sirviese para todos los objetos de la jerarquía que tiene como raíz a la clase *PuestoTrabajo*, evitando tener que crear una función de este tipo para cada una de las clases derivadas. Con este fin, pretendemos usar el polimorfismo para que el argumento *trabajador* de la función *diaLaborable* se comporte, unas veces, como un objeto de la clase base, y otras, como uno de la clase derivada, como se aprecia en la función **main**.

```
class PuestoTrabajo {
public:
  void trabajar(){
    cout << "Trabajo" << endl;
  }
};
class Profesor : public PuestoTrabajo {
public:
  //Redefinición
  void trabajar(){
    cout << "Enseñando a los alumnos" << endl;
  }
};
//Polimorfismo y extensibilidad
void diaLaborable(PuestoTrabajo &trabajador) {
  trabajador.trabajar();
}
void main() {
```

```
    PuestoTrabajo puesto;
    Profesor profe;
    //Con referencias
    diaLaborable(puesto);
    diaLaborable (profe);
}
```

Visto el código de nuestro ejemplo, pasemos a analizar el resultado mostrado por pantalla. Como se observa, no estamos logrando nuestro objetivo: siempre se está invocando la función *trabajar* de la clase base, independientemente del tipo del objeto que pasemos en las llamadas a la función *diaLaborable*. Por lo tanto, no estamos consiguiendo el comportamiento polimórfico que deseábamos.

```
Trabajo
Trabajo
```

¿Por qué se produce este comportamiento? Para responder a esta pregunta es necesario explicar el concepto de **ligadura** o **call binding.**

4.2.1. Ligadura

La ligadura se refiere al proceso en el que el compilador relaciona cada llamada a una función con la dirección de memoria en la que se encuentra el cuerpo de la misma dentro del segmento de texto. Por ejemplo, las llamadas a la función *diaLaborable* desde la función main serán relacionadas con la posición que ocupe en memoria dicha función, de manera que en la ejecución se ejecute el cuerpo de esta función y no el de ninguna otra. Por lo tanto, básicamente, debemos quedarnos con la idea de que la ligadura es el proceso de asociar un cuerpo de función a cada llamada de función que aparezca en el programa.

Este proceso de ligadura se puede realizar de dos formas distintas: estática o dinámicamente. La **ligadura estática o temprana** se realiza en tiempo de compilación. Ésta es la única opción que existe en los lenguajes procedurales. En este caso, el compilador se encarga de resolver las llamadas a funciones. Evidentemente, en el caso de usar ligadura estática en los lenguajes orientados a objetos, las llamadas a las funciones miembro de un objeto se resolverán usando como referencia la clase de ese objeto. Esto es justamente lo que está sucediendo en nuestro ejemplo anterior, en el que las llamadas a la función *trabajar* se resuelven en tiempo de compilación, usando la clase del objeto sobre el que se invoca esta función. Es decir, usando la clase *PuestoTrabajo*, que es la clase del argumento *trabajador* declarado para la función *diaLaborable*. Esta es la razón por la que no conseguimos el comportamiento polimórfico esperado.

Por otra parte, existe la **ligadura dinámica o tardía**, que se produce en tiempo de ejecución. Este tipo de ligadura es propia de los lenguajes orientados a objetos. Mediante esta ligadura las llamadas a funciones se resuelven en función de la clase efectiva del objeto y no de la clase declarada para la referencia

a ese objeto. En el caso del ejemplo anterior, esto se traduce en que la ligadura de la llamada a la función *trabajar* se resuelve en función de la clase del objeto que se le pasa por argumento a la función *diaLaborable* y no por la clase del argumento *trabajador*.

4.2.2. Funciones virtuales

En el lenguaje C++, la ligadura usada por defecto es la ligadura estática. Sin embargo, podemos indicar para qué funciones de una clase se deben resolver sus llamadas mediante ligadura dinámica. Para esta labor, se introduce la palabra reservada **virtual**. Veamos cómo se logra la ligadura dinámica en el ejemplo anterior.

```
class PuestoTrabajo {
public:
  virtual void trabajar(){
    cout << "Trabajo" << endl;
  }
};
```

Como vemos en el ejemplo, para indicar que queremos la ligadura dinámica de una función sólo es necesario especificarlo en la clase base. Se puede volver a indicar en las clases derivadas, pero no tiene ningún efecto; ya que, una vez declarada una función como virtual en una clase, esa función será virtual para todas las clases derivadas de esa clase. Veamos, ahora, la salida que genera nuestro programa.

```
Trabajo
Enseñando a los alumnos
```

Ahora, sí funciona como deseábamos. Cuando a la función *diaLaborable* le pasamos un objeto de la clase base, se invoca a la función *trabajar* de la clase *PuestoTrabajo*. Mientras que si le pasamos un objeto de la clase derivada, se invoca a la función *trabajar* de la clase *Profesor*.

Antes de seguir, es necesario reflexionar sobre el hecho de que la ligadura dinámica no sea la opción por defecto en C++ con la importancia que tiene el polimorfismo para los LOO. Para resolver esta dicotomía es necesario ser consciente de que la ligadura tardía no se produce por arte de magia, sino que se debe a una serie de estructuras y lógica adicionales que el compilador introduce en el código de nuestro programa, con el consiguiente efecto negativo en el rendimiento del programa. Por otra parte, el lenguaje C++ fue diseñado siguiendo la siguiente premisa de B. Strouptrup: "If you don´t need it, don´t pay for it".

En conclusión, si sólo vamos a usar la ligadura tardía en determinadas funciones de nuestro programa, no merece pagar el coste para todas las funciones,

sino sólo para las que sea necesario. Esta es la razón por la que la ligadura dinámica no es la opción por defecto en el lenguaje C++.

4.2.3. Sobrescritura

Cuando vimos la herencia, se comentó el concepto de redefinición. Una clase derivada podía proporcionar una nueva definición para una función ya definida en la clase base. En el caso de funciones virtuales, se habla de sobrescritura en lugar de redefinición. Una clase derivada puede sobrescribir una función virtual de la clase base. En el ejemplo anterior, la función *trabajar* de la clase Profesor sobrescribe la función virtual *trabajar* de la clase *PuestoTrabajo*.

4.2.3.1. Extensibilidad y estructuras polimórficas

Como beneficio directo del uso del polimorfismo logramos mejorar la extensibilidad de nuestro programa. Ahora, siguiendo el ejemplo, es posible crear nuevas clases derivadas de la clase *PuestoTrabajo* sin tener que modificar para nada las funciones que manipulan objetos de estas clases, como la función *diaLaborable*. Por ejemplo, podemos crear la clase *Pastelero* y llamar a la función *diaLaborable* con un objeto de esta clase sin tener que modificar esa función.

```
class Pastelero: public PuestoTrabajo {
public:
  void trabajar(){
    cout << "Preparando pasteles" << endl;
  }
};
void main(){
  ...
  Pastelero paco;
  diaLaborable(paco);
  ...
}
```

Por otra parte, otra característica interesante es la posibilidad de crear estructuras de datos polimórficas. De esta forma podemos lograr un código simple pero muy poderoso. En el siguiente ejemplo, hemos creado un array polimórfico de trabajadores, de manera que cada posición del array está ocupada por una clase distinta de trabajador. Y, además, conseguimos poner a trabajar a todos los trabajadores con sólo dos líneas de código, aprovechando al máximo el polimorfismo.

```
void main(){
  ...
  PuestoTrabajo *trabajadores[3];
```

```
  ...
  trabajadores[0] = new Profesor;
  trabajadores[1] = new Pastelero;
  trabajadores[2] = new Medico;
  ...
  for (i=0; i<3; i++)
    trabajadores[i]->trabajar();
  ...
}
```

Apuntar, de nuevo, que para declarar el array polimórfico usamos como tipo la clase base. En este caso, estamos usando punteros a objetos porque, como ya dijimos, el polimorfismo en C++ se logra con direcciones a objetos.

4.2.3.2. Método virtual no sobrescrito

Hasta ahora, en todos los ejemplos que hemos visto, la clase derivada sobrescribía las funciones declaradas virtuales en la clase base. Por ejemplo, ocurría en las clase *PuestoTrabajo* y sus derivadas con la función *trabajar*. Sin embargo, ¿qué ocurriría si una clase derivada no sobrescribe una de las funciones virtuales? En el siguiente ejemplo, se introduce una nueva función virtual *descansar* y una nueva clase derivada *ProfesorVisitante* a partir de la clase *Profesor*.

```
class PuestoTrabajo {
public:
  virtual void trabajar(){
    cout << "Trabajando" << endl;
  }
  virtual void descansar(){
    cout << "Descansando" << endl;
  }
};
class Profesor : public PuestoTrabajo {
public:
  //Sobrescritura
  void trabajar(){
    cout << "Enseñando a los alumnos" << endl;
  }
  void descansar(){
    cout << "Tomando café" << endl;
  }
};
class ProfesorVisitante: public Profesor {
public:
  void trabajar(){
```

```
    cout << "Dando coferencias" << endl;
  }
};
void diaLaborable(PuestoTrabajo &trabajador) {
  trabajador.trabajar();
  trabajador.descansar();
}
void main() {
  ...
  ProfesorVisitante tommas;
  diaLaborable(tommas);
  ...
}
```

En este caso, la clase *ProfesorVisitante* no sobrescribe la función virtual descansar. Por lo tanto, cuando se llama a esta función desde un objeto de esta clase, en realidad se está invocando a la función *descansar* de su clase base, es decir, a la de *Profesor*, como se puede comprobar en la salida del programa.

```
Dando conferencias
Tomando café
```

El comportamiento del compilador C++, en este caso, consiste en buscar en la clase del objeto la función invocada, si no la encuentra sube a la clase inmediatamente superior y la busca allí, continuando este algoritmo hasta que encuentre un cuerpo de función al que asociar la llamada. En el ejemplo, si la clase *Profesor* no hubiese rescrito la función *descansar*, se hubiese ejecutado la de la clase *PuestoTrabajo*.

4.2.4. Clases abstractas

Generalmente, cuando se construye una jerarquía de clases, suele ocurrir que la clase raíz no posee una funcionalidad real, sino que más bien su función principal consiste en declarar una interfaz funcional común a todas sus clases derivadas. Por lo tanto, crear un objeto de esa clase no tiene ninguna utilidad. Y tampoco sirve para nada definir sus funciones, basta con declararlas.

En los LOO, este tipo de clases se denominan **clases abstractas**. Las clases abstractas se denominan así porque contienen funciones no definidas, es decir, que no tienen cuerpo. Además, son clases que no se pueden instanciar, no se puede crear un objeto de esa clase. Su única utilidad consiste en mantener la interfaz funcional común a todas las clases derivadas.

En el lenguaje C++, una clase es abstracta cuando tiene alguna función virtual pura. Una función virtual pura es una función no definida. Veamos cómo se declara una función de este tipo en un ejemplo. Si analizamos el ejemplo que

estamos usando en este capítulo, podremos comprobar que la clase *PuestoTrabajo* debería ser una clase abstracta; ya que, sus funciones no tienen ninguna funcionalidad real. Por lo tanto, podemos declarar esta clase como abstracta. El resto de clases no se muestran porque no sufren ningún cambio.

```
class PuestoTrabajo { //Clase abstracta
public:
  virtual void tocar()=0; //Función virtual pura
  virtual void ajustar()=0; //Función virtual pura
};
void main() {
  ...
  Profesor ana;
  diaLaborable(ana);
  ...
  PuestoTrabajo * profe1; //punteros a clase abstracta
  PuestoTrabajo & profe2; //referencias a clase abstracta
  ...
  PuestoTrabajo profe3; //No objetos de clase abstracta
  ...
}
```

Evidentemente, si una clase derivada de una clase abstracta no proporciona una definición para alguna de las funciones virtuales puras, la clase derivada seguirá siendo una clase abstracta. Por lo tanto, tampoco se podrá crear ningún objeto de la misma.

4.2.5. Constructores y funciones virtuales

Los constructores, como se vio en el capítulo 2, son unas funciones miembro especiales que se invocan cuando se crea un objeto de una clase. En el capítulo 3, por su parte, se realizó una prolija exposición del comportamiento de los constructores en la herencia. Básicamente, cuando se creaba un objeto de una clase derivada, se invocaba primero el constructor de la clase base y luego el de la clase derivada.

Por otra parte, una vez explicado el concepto de funciones virtuales, ¿qué sucede si desde un constructor se realiza una llamada a una función virtual? Veamos un ejemplo para poder reflexionar sobre esta cuestión.

```
class Profesor: public PuestoTrabajo {
public:
  //Uso de funciones virtuales en constructor
  Profesor(){
    cout << "Creando profesor" << endl;
    trabajar();
```

```
  }
  virtual void trabajar(){
    cout << "Enseñando a los alumnos" << endl;
  }
  virtual void descansar(){
    cout << "Tomando café" << endl;
  }
};
class ProfesorVisitante : public Profesor {
  int costeHora;
public:
  ProfesorVisitante(){
    cout << "Creando profesor visitante" << endl;
    costeHora = 100;
  }
  void trabajar(){
    int costeTotal = costeHora * 2;
    cout << "Dando conferencias" << endl;
    cout << "Coste conferencia = "<< costeTotal <<endl;
  }
};
void main() {
  ...
  ProfesorVisitante tommas;
  ...
}
```

En este caso, tenemos dos clases: *Profesor* y *ProfesorVisitante*. La clase *Profesor* declara como virtuales las funciones *trabajar* y *descansar*. Y la clase *ProfesorVisitante* sobrescribe la función virtual *trabajar*. Como se puede apreciar, el constructor de la clase *Profesor* contiene una llamada a la función virtual *trabajar*. Cuando se crea un objeto de la clase *Profesor*, el constructor invocará la función *trabajar* definida en esa misma clase. Sin embargo, pensemos qué sucederá cuándo se cree el objeto *tommas*. Este objeto es de la clase *ProfesorVisitante* que deriva de la clase *Profesor*; por lo tanto, primero se invoca al constructor de la clase *Profesor*. Como hemos visto, el constructor de la clase Profesor contiene una llamada a la función virtual *trabajar*. Pero, ¿qué función se invoca: la de *Profesor* o la de *ProfesorVisitante*? Según el concepto de funciones virtuales, se debe invocar a la función perteneciente a la clase del objeto que se está creando, en este caso, a la definida en *ProfesorVisitante*. No obstante, la función *trabajar* de *ProfesorVisitante* está usando un atributo miembro de esa clase, pero ese atributo todavía no ha sido correctamente inicializado porque todavía no se ha ejecutado el constructor de la clase *ProfesorVisitante*, estamos todavía en el constructor de la clase *Profesor*. Por lo tanto, para evitar

este tipo de problemas, en realidad se invocará a la función *trabajar* de la clase *Profesor*.

En resumen, si aparece en un constructor una llamada a una función virtual, el mecanismo virtual no funcionará y se invocará la versión local (para ese constructor) de la función virtual.

4.2.6. Destructores

4.2.6.1. Destructor virtual

Como ya vimos en capítulos anteriores, los destructores son también funciones miembro especiales presentes en todas las clases que se invocan cada vez que se elimina un objeto. En la herencia, también poseen un comportamiento particular. Cuando se crea un objeto de una clase derivada se invocan sus destructores en secuencia desde la clase derivada hasta la base, justamente en orden inverso a los constructores.

El uso del polimorfismo tiene consecuencias sobre el comportamiento de los constructores de las que debemos ser conscientes. En primer lugar, cuando usamos el polimorfismo estamos asignando un objeto de una clase derivada a la dirección de un objeto de una clase base. Por lo tanto, ¿qué pasa cuando se intenta eliminar el objeto apuntado esa dirección? Consideremos el siguiente ejemplo:

```
class PuestoTrabajo {
public:
  virtual void trabajar(){
    cout << "Trabajando" << endl;
  }

  //...

  ~PuestoTrabajo(){
    cout << "Eliminando" << endl;
  }
};
class Profesor : public PuestoTrabajo {
public:
  void trabajar(){
    cout << "Enseñando a los alumnos" << endl;
  }

  //...
  ~Profesor(){
    cout << "Eliminando profesor" << endl;
  }
```

```
};
void main() {
  PuestoTrabajo* pt = new Profesor;
  pt->trabajar();
  delete pt;
}
```

La salida de este ejemplo será:

```
Enseñando a los alumnos
Eliminando
```

En este ejemplo, tenemos dos clases: *PuestoTrabajo* y *Profesor*. Cada una de éllas tiene su propio destructor. Y en la función **main** estamos haciendo uso del polimorfismo creando un puntero a un objeto de la clase *Profesor*, que asignamos a un puntero de la clase *PuestoTrabajo*. Cuando decidimos eliminar el objeto al que apunta el puntero *pt*, la idea que tenemos es que destruya el objeto de la clase *Profesor*. Por lo tanto, se debería invocar, primero, el destructor de la clase *Profesor* y, luego, el de la clase *PuestoTrabajo*. Sin embargo, si comprobamos la salida de nuestro programa, vemos que *pt* se comporta como un objeto de la clase *Profesor* cuando se invoca la función *trabajar*, pero que cuando se elimina sólo se ejecuta el destructor de la clase base.

En este caso estamos obteniendo un comportamiento erróneo. Para lograr un funcionamiento correcto, es necesario declarar el destructor como virtual. De esta manera, conseguimos que se respete la secuencia correcta de llamadas a los destructores. En nuestro ejemplo, bastaría entonces con modificar la declaración del destructor en la clase base.

```
class PuestoTrabajo {
public:
  virtual void trabajar(){
    cout << "Trabajando" << endl;
  }
  //...
  virtual ~PuestoTrabajo(){
    cout << "Eliminando" << endl;
  }
};
```

La salida de este ejemplo será:

```
Enseñando a los alumnos
Eliminando profesor
Eliminando
```

Uno de los errores más molestos de programación en C++ es olvidar declarar el destructor de la clase base como virtual, porque no suele ocasionar errores de ejecución pero introduce pérdidas de memoria.

4.2.6.2. Destructor virtual puro

Éste es sin duda un caso curioso de función virtual pura. En C++ podemos declarar el constructor como una función virtual pura; sin embargo, es obligatorio que el destructor posea una definición, un cuerpo. La razón de que deba poseer un cuerpo se fundamenta en el comportamiento de los destructores en la herencia. El destructor es una función especial que se invoca en secuencia desde la clase derivada hasta la base; por lo tanto, es necesario un cuerpo para soportar este comportamiento. Pero, entonces, ¿para qué sirve declararlo como una función virtual pura? Pues simplemente para conseguir que la clase base sea abstracta, en el caso de que ninguna de sus otras funciones miembro sean virtuales puras. Siguiendo con nuestro ejemplo, el código quedaría así:

```
class PuestoTrabajo {
public:
  virtual void trabajar(){
    cout << "Trabajando" << endl;
  }
  //...

  virtual ~PuestoTrabajo()=0;
};
PuestoTrabajo::~PuestoTrabajo() {
  cout << "Eliminando" << endl;
}
class Profesor : public PuestoTrabajo {
public:
  void trabajar(){
    cout << "Enseñando a los alumnos" << endl;
  }
};
```

Como se aprecia en el ejemplo, no es obligatorio redefinir un destructor virtual puro en las clases derivadas; ya que, el compilador siempre crea un destructor. Por lo tanto, la clase derivada no será abstracta.

4.2.6.3. Destructores y funciones virtuales

Al igual que pasaba con los destructores, tampoco funciona el mecanismo virtual en las llamadas a funciones virtuales desde los destructores. Siempre se usa la versión local de la función; ya que, pueden haber sido destruidos miembros que puedan ser accedidos desde una función virtual más "derivada" dado el orden de invocación de los destructores.

```
class Profesor: public PuestoTrabajo {
```

```
public:
  //Uso de funciones virtuales en constructor
  Profesor(){
    cout << "Creando profesor" << endl;
  }
  virtual void trabajar(){
    cout << "Enseñando a los alumnos" << endl;
  }
  virtual void descansar(){
    cout << "Tomando café" << endl;
  }
  virtual ~Profesor(){
    cout << "Eliminando" << endl;
    descansar();
  }
};
class ProfesorVisitante : public Profesor {
  char *paisOrigen;
public:
  ProfesorVisitante(){
    cout << "Creando profesor visitante" << endl;
    paisOrigen = new char[10];
    strcpy(paisOrigen,"Argentina");
  }
  ...
  void descansar(){
    cout << "Hablar de "<< paisOrigen <<endl;
  }

  ~ProfesorVisitante(){
    cout << "Eliminando profesor visitante" << endl;
    delete [] paisOrigen;
  }
};
void main() {
  ...
  ProfesorVisitante tommas;
  ...
}
```

En este caso, tenemos dos clases: *Profesor* y *ProfesorVisitante*. La clase *Profesor* declara como virtuales las funciones *trabajar* y *descansar*. Y la clase *ProfesorVisitante* sobrescribe la función virtual *descansar*. Como se puede apreciar, el destructor de la clase *Profesor* contiene una llamada a la función virtual *des-*

cansar. Cuando se elimina un objeto de la clase *Profesor*, el destructor invocará la función *descansar* definida en esa misma clase. Sin embargo, pensemos qué sucederá cuándo se elimine el objeto *tommas*. Este objeto es de la clase *ProfesorVisitante* que deriva de la clase *Profesor*; por lo tanto, primero se invoca al destructor de la clase *ProfesorVisitante*. Como hemos visto, el destructor de la clase *Profesor* contiene una llamada a la función virtual *descansar*. Pero, ¿qué función se invoca: la de *Profesor* o la de *ProfesorVisitante*? Según el concepto de funciones virtuales, se debe invocar la función perteneciente a la clase del objeto que se está eliminando, en este caso, a la definida en *ProfesorVisitante*. No obstante, la función *descansar* de *ProfesorVisitante* está usando un atributo miembro de esa clase, *paisOrigen*, pero ese atributo ya ha sido eliminado porque ya se ha ejecutado el destructor de la clase *ProfesorVisitante*. Por lo tanto, para evitar este tipo de problemas, en realidad se invocará a la función descansar de la clase *Profesor*.

En resumen, si aparece en un destructor una llamada a una función virtual, el mecanismo virtual no funcionará y se invocará la versión local (para ese destructor) de la función virtual.

4.3. Sobrecarga y sobrescritura

La sobrecarga de funciones miembro tenía un comportamiento característico cuando aparecía entremezclado con el mecanismo de la herencia. En el capítulo 3, se explicó cómo se comportaban la sobrecarga y la redefinición cuando aparecían juntas. Una vez explicado el polimorfismo, es necesario volver a hacer esta reflexión desde el punto de vista de las funciones virtuales. En este caso, vamos a comprobar qué sucede cuando mezclamos sobrecarga y sobrescritura de funciones miembro. Con este fin, vamos a usar el mismo ejemplo que vimos en el capítulo 3, pero con funciones virtuales.

```
class Documento{
public:
  virtual int archivar(){
    cout<<"Documento archivado"<<endl;
    return 1;
  }
  virtual int archivar(int estanteria){
    cout<<"Documento archivado en "<<estanteria<<endl;
    return 1;
  }
};
class Libro: public Documento{
public:
  //Sobrescritura de una función virtual sobrecargada
```

```
  int archivar(){
    cout<<"Libro archivado"<<endl;
    return 1;
  }
};
class Articulo: public Documento{
public:
  //Nueva versión de una función virtual: cambio parámetros
  int archivar(char *clave){
    cout<<"Articulo archivado por "<<clave<<endl;
    return 1;
  }
};
class Informe: public Documento{
public:
  //Nueva versión de una función virtual: cambio retorno
  //Error de compilacion
  void archivar(){
    cout<<"Informe archivado" <<endl;
  }
};
```

En este caso tenemos la clase base, *Documento*, y tres clases que derivan de ella: *Libro*, *Articulo* e *Informe*. La clase *Documento* define dos versiones de la función virtual *archivar*, es decir, esta función aparece sobrecargada. Por su parte, la clase *Libro* sobrescribe una de las versiones de la función *archivar*. Mientras que las otras dos clases proporcionan una nueva versión de esa función: la primera, variando el número de parámetros y, la segunda, el tipo de retorno. En este caso, el compilador no permite que se cambie el tipo de retorno en la clase *Informe*; puesto que se debe asegurar de que puedas llamar a la función *archivar* polimórficamente desde una dirección a un objeto de la clase *Documento*, en la que se espera que esa función devuelva un valor de tipo **int**. Ahora, la cuestión consiste en saber qué versiones de la función *archivar* están disponibles para los objetos de las otras dos clases derivadas.

```
int main() {
  int x;
  Libro libro;
  x = libro.archivar();
  x = libro.archivar(6);  //error: versión no disponible
  Articulo articulo;
  x = articulo.archivar();   //error: versión no disponible
  x = articulo.archivar("A");
  Documento& documento = articulo; //upcasting
```

```
    x = documento.archivar();
    x = documento.archivar(6);
    x = documento.archivar("A");  //error: versión no disponible
};
```

En el caso del objeto de la clase *Libro*, la sobrescritura de una de las versiones de la función *archivar* supone que la otra versión pase a ser inaccesible. Por otro lado, como muestran las clases *Articulo*, si la clase derivada define una nueva versión de la función *archivar*, cambiando la lista de parámetros, las versiones definidas en la clase base se hacen inaccesibles. Sin embargo, si asignamos el objeto *Articulo* a uno de la clase base (**upcasting**), están accesibles las versiones definidas en la clase base, pero no lo están las definidas en la clase derivada *Articulo*.

4.4. Resumen

Como hemos podido ver, el polimorfismo está estrechamente relacionado con la abstracción de datos y la herencia. El polimorfismo aumenta enormemente el poder de expresividad del lenguaje, favoreciendo la extensión del código. Básicamente, permite escribir código con un comportamiento dinámico. Este código presenta un comportamiento diferente según el tipo de objeto que está en ejecución con independencia del tipo de objeto que se especificó en tiempo de compilación.

En definitiva, mediante el polimorfismo obtenemos un grado más de reutilización de código. Si mediante la herencia reutilizábamos código de una clase para construir otras, mediante el polimorfismo reutilizamos el código de funciones que trabajan con un objeto de la clase base para que trabaje con objetos de las clases derivadas. Este comportamiento se conoce como *el principipio de la habilidad de sustitución* y supone que el código escrito para un objeto de un tipo no necesita ser modificado si se sustituye ese objeto por otro de un subtipo del mismo.

En C++, este comportamiento se logra a partir de las funciones virtuales. El mecanismo de funciones virtuales permite que una clase derivada sobrescriba los métodos que hereda y que las llamadas a los mismos se resuelvan mediante ligadura tardía o dinámica. Sin embargo, éste no es el comportamiento por defecto en C++. Puede parecer una contradicción el hecho de que un lenguaje que soporte el paradigma de orientación a objetos no tenga como opción por defecto el polimorfismo: pilar básico de la POO. No obstante, como ya se ha comentado, el lenguaje C++ es un lenguaje multiparadigma que intenta satisfacer requisitos dispares y contradictorios, a veces. Por esta causa y por el principio de eficiencia que rigió el diseño de este lenguaje se decide que las funciones vituales no sean la opción por defecto. Para comprender en toda su profundidad esta decisión, se anima al lector a consultar la bibliografía de este libro en busca de una explicación más detallada.

Capítulo 5

Excepciones

5.1. Concepto

Una excepción es una situación anormal que se produce en un programa en tiempo de ejecución, por ejemplo, un error. Actualmente, los lenguajes suelen proporcionar mecanismos para el manejo de excepciones, C++ es uno de estos lenguajes. Aunque el manejo de errores con excepciones no es un concepto referente a la POO, es conveniente estudiarlo debido a su utilidad en ocasiones en las que debamos manejar cierto tipo de excepciones de una forma más estructurada. El manejo de excepciones es un mecanismo que permite manejar los errores en tiempo de ejecución de una manera ordenada y controlada. En concreto, este mecanismo fue ideado para dar soporte al tratamiento de errores en programas formados por componentes desarrollados independientemente, por ejemplo, bibliotecas. Este mecanismo permite que la situación excepcional no tenga que ser solucionada en la función donde surgió, sino que pueda ser enviada a un contexto superior (función invocante) en el que exista suficiente información para resolver dicha situación. Pensemos, por ejemplo, que hemos implementado un programa que usa una biblioteca, desarrollada por otro programador, que nos gestiona el acceso a memoria dinámica. Si se produce un error de falta de memoria dentro de una de las funciones de la biblioteca, la función de la biblioteca no puede dar una solución a este error que sirva para todos los programas que usen esta biblioteca. Por lo tanto, la biblioteca debe pasar ese error al programa que invoque una función de la biblioteca para que el programador que usa esa biblioteca pueda decidir que se hace en esa situación.

5.2. Manejo de errores tradicional

El principal inconveniente que presentaba el manejo de errores antes de que existiera ningún manejador de excepciones era que los errores debían se tratados

de una forma *manual*. Algunas de las alternativas utilizadas más usualmente para tratar excepciones, fallos o anomalías en programas eran las siguientes:

- Terminación del programa. Se utilizaba esta solución cuando no había posibilidad de recuperación del error o ésta era muy difícil. Un ejemplo de esta opción es el utilizado en la instrucción **assert**, que se mostraba en el capítulo 2, cuando, si se producía un error al reservar memoria para una variable en el constructor de la clase *Telefono*, se producía la finalización inmediata del programa:

```
int tama=0; //inicializamos el campo número
tama=strlen(_numero);
numero=new char[tama+1];
assert(numero); //comprueba memoria
strcpy(numero, _numero);
```

- Control del valor devuelto por una función, indicando si finaliza la función de forma correcta o incorrecta con dicho valor. Por ejemplo, una función podría devolver 0 si ha finalizado con éxito o 1 si ha finalizado con algún tipo de error. También se puede utilizar la instrucción **exit** con la misma finalidad, es decir, indicar el valor con el cual se sale del programa (de forma correcta o incorrecta). Por ejemplo, supongamos que un programa debe cumplir unas condiciones iniciales para poder continuar realizando ciertos cálculos, en caso contrario, el programa tendría que detenerse. Podría realizarse de la siguiente forma:

```
int main() {
  if (condiciones_iniciales()) {
    realizar_calculos( );
    return (0);
  }
  cout<<"No se cumplen las condiciones iniciales";
  exit(1);
}
```

- Crear una interrupción o señal. Esta señal es una interrupción que solicita al sistema que llame a un manejador de señales. Mediante la función C **raise(int sig)** se envía la excepción especificada mediante el parámetro sig al programa que se está ejecutando actualmente. Estas excepciones se manejan mediante la función **signal**. Esta función tiene el siguiente formato:

```
void (*signal (int sig, void (*funcsig)(int))(int)
```

- Otra opción sería crear una llamada a la función **abort** que produciría una terminación anormal del programa. Devuelve al programa que realiza

la llamada un valor de 3. Esta función suele utilizarse para evitar que un programa fuera de control cierre los archivos activos, ya que al salir con esta función no se limpian los búferes de archivos.

```
int main () {
  if existe_error( ) abort();
  return (0); //salida normal del programa
}
```

La mayor ventaja a la hora de manejar excepciones es que un programa puede llamar a una rutina de tratamiento de errores justo en el momento en que se produzca algún tipo de error mediante una estructura de control específica, como puede ser el **bloque-try**.

5.3. Excepciones en C++

El lenguaje C++, a partir de la versión 3.0 de AT&T y el borrador de ANSI/ISO C++, soporta el mecanismo de manejo de excepciones. Una vez producida una excepción en un programa podríamos finalizar el programa, sin embargo si se provoca el lanzamiento de una excepción podríamos conocer el motivo que ha provocado dicha excepción y podríamos también realizar ciertas acciones antes de finalizar el programa. Veremos pues a continuación el funcionamiento del mecanismo de excepciones.

5.3.1. Lanzar y capturar excepciones

El manejo de excepciones en C++ se basa en tres instrucciones, éstas son **try**, **catch** y **throw**. Se puede resumir el manejo de excepciones de una forma muy general: Las instrucciones que se utilizarán para monitorizar excepciones se sitúan dentro de un bucle **try**. Si se produce alguna excepción o error dentro de un bloque **try** se notifica mediante **throw**. Para capturar la excepción se utiliza la instrucción **catch** y se procesa la excepción. Por lo tanto, cualquier instrucción que pueda notificar una excepción estará dentro de un bloque **try**, la excepción se capturará dentro de un bloque **catch**, que debe encontrarse justo detrás del bloque **try** que ha notificado la excepción. El formato de estas dos instrucciones es el siguiente:

```
try { //bloque de instrucción try
} catch (tipo1 argumento) { //bloque de instrucción catch }
...
catch (tipoN argumento) { //bloque de instrucción catch }
```

Cuando se notifique una excepción, esta será capturada por un bloque **catch**. Como se ve en la definición, puede haber más de una instrucción **catch** asociada

con una instrucción **try**. Cuando se produce una excepción se ejecutará la instrucción **catch** cuyo tipo coincida con la excepción, ignorando las demás instrucciones **catch**. La variable argumento recibirá el valor de la instrucción. Si no se notifica ninguna excepción no se ejecutará ninguna instrucción **catch**. Para lanzar una excepción se utiliza la sentencia **throw**. La sintaxis de esta sentencia es:

```
throw;
throw expresión;
```

La excepción que se lanza es el valor de la expresión **throw**. Esta expresión se utiliza para inicializar el valor utilizado en la instrucción **catch** o lo que es lo mismo, el valor de la variable argumento de la instrucción **catch**. El primer formato indica que la excepción debe lanzarse de nuevo. Debe existir realmente una excepción o de lo contrario se llamaría a la función **terminate**. El segundo formato indica qué expresión se pasa al manejador de excepciones para ser capturada y procesada. Si se notifica una excepción para la que no hay una instrucción **catch** que pueda capturarla, esto provocará una terminación anormal del programa. En este caso se activará una llamada a la función **terminate**. Esta función llamará a la función **abort** y se detendrá la ejecución del programa. Veamos un ejemplo en el que se lanza una excepción:

```
int main() {
  cout <<"Tratando excepciones"<<endl;
  try {
    cout <<"Dentro de la instrucción try"<<endl;
    throw 5; //lanzando una excepción
    cout <<"Esta parte de código no se ejecuta"<<endl;
  } catch (int argumento) {
    cout<<"Capturado el error: "<<argumento<<endl;
  }
  cout <<"Finalizado el tratamiento de excepciones"<<endl;
}
```

La salida por pantalla de este ejemplo será:

```
Tratando excepciones
Dentro de la instrucción try
Capturado el error: 5
Finalizado el tratamiento de excepciones
```

Como se puede ver, dentro del bloque **try** se ejecuta la instrucción **cout** y la instrucción **throw**. Una vez lanzada la excepción, se pasa el control a la instrucción **catch** y termina la instrucción **try**, por lo tanto la instrucción:

```
cout<<"Esta parte de código no se ejecuta"<<endl;
```

no se ejecuta nunca, como se expresa en el mensaje. En este sentido, puede decirse que la instrucción **throw** es como una sentencia **return** con la diferencia de que el control no vuelve a la línea siguiente a la que invocó la función.

Hay que tener en cuenta que el tipo de excepción debe coincidir con el tipo especificado en una de las instrucciones **catch**, si no ocurre esto se producirá una terminación anormal del programa. Veamos este caso con otro ejemplo, modificando el tipo que recibe la instrucción **catch** mostrada anteriormente.

```
int main() {
  cout <<"Tratando excepciones"<<endl;
  try {
    cout <<"Dentro de la instrucción try"<<endl;
    throw 5; //lanzando una excepción
    cout <<"Esta parte de código no se ejecuta"<<endl;
  } catch (double argumento) {
    cout<<"Capturado el error: "<<argumento<<endl;
  }
  cout <<"Finalizado el tratamiento de excepciones"<<endl;
}
```

La salida que generaría este programa sería la siguiente:

```
Tratando excepciones
Dentro de la instrucción try
Abnormal program termination
```

Esta salida es debida a que la excepción lanzada de tipo **int** no es capturada por la sentencia **catch**, ya que ésta es de tipo **double**. Podríamos solucionarlo añadiendo la sentencia **catch** del ejemplo anterior que posee el tipo **int** y obtendríamos el resultado mostrado en el primer programa.

Como vimos anteriormente, una excepción puede lanzarse desde una instrucción que esté fuera del bloque **try**, siempre que esté dentro de una función cuya llamada esté dentro de un bloque **try**. Veamos un ejemplo:

```
void mostrarDatos(char *dato, int longitud) {
  if (longitud < 20)
    cout << "El dato a mostrar es: "<<dato<<endl;
  else throw 0;
}
void main() {
  cout <<"Tratando excepciones dentro de una función"<<endl;
  try {
    cout <<"Dentro de la instrucción try"<<endl;
    mostrarDatos("Ejemplo pequeño", 15);
    mostrarDatos("Ejemplo cadena mayor de 20", 56);
```

```
  } catch (int i) {
    cout<<"Capturado el error: "<<i<<endl;
  }
  cout <<"Finalizado el tratamiento de excepciones"<<endl;
}
```

El resultado que obtendría este programa sería el siguiente:

```
Tratando excepciones dentro de una función
Dentro de la instrucción try
El dato a mostrar es: Ejemplo pequeño
Capturado el error: 0
Finalizado el tratamiento de excepciones
```

Cuando se llama a la función con una cadena cuyo tamaño es menor que 20 caracteres, se llama a la función mostrarDatos, se muestra la cadena por pantalla y se vuelve a la siguiente línea de la instrucción **try** desde donde se llamó a la función *mostrarDatos*, sin embargo, cuando se llama a la función con una cadena cuyo tamaño es mayor que 20 caracteres, se lanza desde la función la excepción con valor cero que es capturada por la instrucción **catch**.

Una instrucción **catch** puede capturar distintos tipos de excepciones, veamos un ejemplo en el que la instrucción catch puede capturar tipos de datos **int** o cadenas de caracteres:

```
void seleccionar(int dato) {
  try {
   cout <<"Dentro de la instrucción try"<<endl;
   if (dato)
     throw dato;
   else throw "Dato menor o igual a 0";
  } catch (int i) {
    cout<<"Capturado el error: "<<i<<endl;
  } catch (char *cad) {
    cout<<"Capturado el error de tipo cadena: "<<cad<<endl;
  }
}
void main() {
  cout <<"Tratando excepciones de varios tipos"<<endl;
  seleccionar(5);
  seleccionar(3);
  seleccionar(0);
  cout <<"Finalizado el tratamiento de excepciones"<<endl;
}
```

El resultado de este programa sería el siguiente:

```
Tratando excepciones de varios tipos
Dentro de la instrucción try
Capturado el error: 5
Dentro de la instrucción try
Capturado el error: 3
Dentro de la instrucción try
Capturado el error de tipo cadena:Dato menor o igual a 0
Finalizado el tratamiento de excepciones de varios tipos
```

A veces puede que necesitemos capturar todas las excepciones, independientemente de su tipo, para realizar esta acción, utilizaremos el siguiente formato de la instrucción **catch**:

```
catch (...) { }
```

Si modificamos el ejemplo anterior con este nuevo formato de **catch**, el programa y su resultado serían los siguientes:

```
void seleccionar(int dato) {
  try {
    cout <<"Dentro de la instrucción try"<<endl;
    if (dato)
      throw dato;
    else
      throw "El dato es menor o igual a 0";
  } catch (...) {
    cout<<"Capturado error"<<endl;
  }
}
void main() {
  cout <<"Tratando excepciones de varios tipos"<<endl;
  seleccionar(5);
  seleccionar(3);
  seleccionar(0);
  cout <<"Finalizado el tratamiento de excepciones"<<endl;
}
```

El resultado de este programa sería el siguiente:

```
Tratando excepciones de varios tipos
Dentro de la instrucción try
Capturado error
Dentro de la instrucción try
Capturado error
Dentro de la instrucción try
Capturado error
Finalizado el tratamiento de excepciones
```

Pueden escribirse en un mismo bloque **try** instrucciones **catch** especificando un tipo de instrucciones **catch** del tipo **catch(...)**. Por ejemplo, las dos instrucciones siguientes podrían escribirse juntas en el ejemplo anterior:

```
try {
  cout <<"Dentro de la instrucción try"<<endl;
  if (dato)
    throw dato;
  else
    throw "Dato menor o igual a 0";
} catch (int i) {
  cout<<"Capturado el error de tipo entero: "<<i<<endl;
} catch (...) {
  cout<<"Capturado error"<<endl;
}
```

En este caso, si se lanza una excepción de tipo **int**, se capturaría con la primera instrucción **catch** y si se lanzara una excepción de cualquier otro tipo, se capturaría con la segunda instrucción **catch**.

5.3.2. Restricción de excepciones permitidas

Pueden restringirse en una función el tipo de excepciones que puede notificar esta función. Para realizar esto, el formato de la función será de una de las siguientes formas:

```
tipo nombre_función (lista parámetros) throw (lista de tipos)
tipo nombre_función (lista parámetros) throw ()
```

La *lista de tipos* serán los tipos de excepciones que la función puede lanzar, si la lista está vacía, la función no podrá lanzar ningún tipo de excepción. Si se intenta lanzar algún tipo de excepción no especificado en la lista de tipos de la función, el programa producirá una llamada a la función **unexpected**, que llamará a su vez a la función **terminate**. Volvamos a ver un ejemplo de la función *seleccionar* mostrada anteriormente, utilizando la restricción de excepciones en dicha función:

```
void seleccionar(int dato) throw (int) {
  if (dato)
    throw dato;
  else
    throw "El dato es menor o igual a 0";
}
void main() { (c == 0)
```

```
  cout <<"Tratando excepciones de varios tipos"<<endl;
  try {
    cout <<"Dentro de la instrucción try"<<endl;
    seleccionar(5);
  } catch (int i) {
    cout<<"Capturado el error: "<<i<<endl;
  } catch (char *cad) {
    cout<<"Capturado el error de tipo cadena: "<<cad<<endl;
  }
  cout <<"Finalizado el tratamiento de excepciones"<<endl;
}
```

En este caso, la función seleccionar sólo puede lanzar excepciones de tipo entero, en este ejemplo, y llamando a la función con el valor 5, el resultado del programa sería el siguiente:

```
Tratando excepciones de varios tipos
Dentro de la instrucción try
Capturado el error: 5
Finalizado el tratamiento de excepciones
```

Si se realizara una llamada a la función seleccionar con un valor 0 *seleccionar(0)*, el programa produciría una terminación incorrecta; ya que, la función intentaría lanzar una excepción de tipo cadena de caracteres y la función no tiene este tipo de excepción dentro de su lista de tipos. Este problema se podría solucionar cambiando la definición de la función seleccionar de la siguiente forma:

```
void seleccionar(int dato) throw (int, char *)
```

Si la función seleccionar fuera de este tipo:

```
void seleccionar(int dato) throw () {
  //líneas que producirían error al ejecutarse
  if (dato)
    throw dato;
  else
    throw "El dato es menor o igual a 0";
}
```

se produciría una terminación incorrecta del programa; ya que esta función no podría lanzar ningún tipo de excepción.

5.3.3. Relanzar excepciones

El manejador de excepciones puede relanzar la excepción que lo activó utilizando la sentencia **throw**. De esta forma, la excepción actual se lanzará a una

secuencia **try/catch** externa. Una de las ventajas de relanzar excepciones es que se permite de esta forma a múltiples controladores acceder a una excepción. Veamos un ejemplo:

```
void relanzar() {
  try {
    throw "mensaje";
  } catch (char *cadena) {
    cout << "Capturado el error en la función: "<<cadena<<endl;
    throw "Mensaje de función relanzar";
  }
}
void main() {
  cout <<"Relanzando excepciones"<<endl;
  try {
    relanzar();
  } catch (char *cad) {
    cout<<"Capturado el error en el main: "<<cad<<endl;
  }
  cout <<"Finalizado relanzar excepciones"<<endl;
}
```

La salida por pantalla será:

```
Relanzando excepciones (c == 0)
Capturado el error en la función: mensaje
Capturado el error en el main: Mensaje de función relanzar
Finalizado relanzar excepciones
```

Se muestra a continuación un ejemplo sencillo de utilización del manejador de excepciones. El ejemplo muestra el lanzamiento de una excepción controlada cuando se intenta reservar memoria para una variable de forma dinámica y no hay memoria disponible, en este caso mostraríamos al usuario un mensaje de error y finalizaría el programa.

```
void main() {
  char *cadena;
  cout <<"Intentando reservar memoria"<<endl;
  try {
    if((cadena=new char[50]) == NULL)
      throw 1;
    else {
      strcpy(cadena, "memoria asignada");
      cout << cadena<<endl;
    }
```

```
  } catch (int i) {
    cout<<"Error al reservar memoria"<<endl;
  }
  cout <<"Finalizada la reserva dinámica de memoria"<<endl;
  delete cadena;
}
```

El resultado del programa en caso de que se reserve la memoria correctamente será el siguiente:

```
Intentando reservar memoria memoria asignada
Finalizada la reserva dinámica de memoria
```

5.3.4. Excepciones definidas por el usuario

El objetivo fundamental de los mecanismos de tratamiento de errores es el paso de información relativa al error desde el punto de detección hasta el punto donde existe suficiente información para resolverlo correctamente. Por lo tanto, si podemos pasar información de ese tipo dentro de la propia excepción, la cantidad de información disponible para tratar el problema se incrementa.

En C++, una excepción realmente es un objeto de una clase que representa un suceso excepcional. El programador puede definir, por lo tanto, la clase de la excepción que se va a elevar en una determinada situación. Evidentemente, como con cualquier clase, se pueden crear jerarquías de clases que representan excepciones, dando lugar a manejadores interesados en un conjunto de excepciones.

En el siguiente ejemplo se muestra la definición de una clase que representa una excepción.

```
class ErrorES
{
  string nombre;
public:
  ErrorES(string fichero){nombre = fichero;}
  void informa(){
    cerr<<''Error E/S (``<<nombre<<'')\n'';
  }
};
void leerFichero(string nombre){
  //...
  throw ErrorES(nombre);
 //...
}
void procesar(){
  try{
```

```
      leerFichero(''uno'');
      leerFichero(''dos'');
      leerFichero(''tres'');
    }
    catch(ErrorES& e)
    {
     //...
     e.informa();
    }
  }
```

En este ejemplo, hemos creado la clase *ErrorES* para representar un error de entrada/salida. Esta clase guarda como atributo el nombre del fichero en el que se ha producido el error. De esta manera, la función *leerFichero* lanza un objeto de esta clase indicando el nombre del fichero que se estaba procesando. Así, la función *procesar*, cuando captura una excepción de este tipo, puede aportar el nombre del fichero que ha presentado problemas, pudiendo tomar las acciones necesarias para solventar este error.

5.3.5. Excepciones en constructores y destructores

Como se ha visto en capítulos anteriores, los **constructores** son funciones miembro especiales que siempre devuelven un objeto de su clase. El problema surge cuando se produce un error en el código del constructor como, por ejemplo, que no se pueda inicializar algún recurso necesario para el objeto. En esta situación es necesario comunicar a la función invocante que se ha producido un error y que el objeto devuelto no es correcto. Sin el mecanismo de tratamiento de excepciones, las soluciones tradicionales han ideado soluciones bastante artificiales como, por ejemplo:

- Devolver un objeto en un estado erróneo y confiar que el usuario (función invocante) compruebe el estado.
- Cambiar el valor de alguna variable no local y confiar que el usuario compruebe dicha variable.
- No realizar ninguna iniciación en el constructor y confiar que el usuario llame a otra función de iniciación antes de usar el objeto.
- Marcar el objeto como "no iniciado" de manera que la primera función invocada sobre el mismo lleve a cabo la iniciación y notifique los posibles errores que se puedan dar.

Estas soluciones complican tanto la codificación de clases como la creación de objetos de las mismas. Parece evidente que la solución pasa por transmitir el error fuera del propio constructor, pero sin complicar ni el código de la

clase ni el código en el que se crean objetos. El mecanismo de tratamiento de excepciones encaja perfectamente con estos requisitos. Por lo tanto, la mejor manera de informar al usuario de que se ha producido un error al crear un objeto es elevando una excepción desde el constructor.

Veamos un ejemplo sencillo del tratamiento de un error detectado en el constructor de una clase.

```
class punto3D
{
  int x, y, z;
  class ptoOrigen{};

public:
  punto3D(int a, int b, int c) {
    if ((a == 0)&& (b == 0) &&(c == 0) )
      throw ptoOrigen();
    x=a;
    y=b;
    z=c;
  }
 };
int main(void){
  try
  {
   punto3D punto(0,0,0);
  }
  catch(punto3D::ptoOrigen)
  {
   //...
  }
}
```

En este ejemplo, se implementa la clase *punto3D*. El constructor parametrizado de esta clase eleva la excepción *ptoOrigen* cuando se intenta crear un punto en el origen de coordenadas. Esta excepción se transmite a la función **main** donde se intenta crear un objeto *punto3D* en el origen de coordenadas y es capturada dentro del **bloque-try**.

En el caso de los **destructores**, el destructor debe evitar elevar excepciones, por norma general. Por ejemplo, en el caso de que se invoque desde el destructor a una función susceptible de elevar excepciones, se deben capturar las mismas mediante un **bloque-try** dentro del destructor.

5.4. Resumen

En este capítulo se ha presentado el tratamiento de excepciones en el lenguaje C++. Las excepciones pueden definirse como situaciones anormales que pueden producirse en algún momento en cualquier programa que se esté ejecutando. En primer lugar se ha hecho una breve introducción sobre las distintas formas en las que se realizaba el tratamiento de excepciones de forma tradicional para, a continuación presentar la forma actual en que podemos lanzar y capturar excepciones utilizando el lenguaje C++, especificando incluso en algunos casos el tipo de excepción. Para finalizar, y como enlace a la POO, en este capítulo se ha mostrado el tratamiento de excepciones en los constructores y destructores de objetos; su principal misión es controlar las posibles excepciones producidas al crear o destruir objetos.

Capítulo 6

Estructuras de datos

6.1. Concepto

Podría definirse una estructura de datos como la unión de un conjunto de variables (pudiendo ser éstas de distinto tipo) que están relacionadas entre sí y las operaciones definidas sobre ese conjunto de variables. Podemos encontrar ejemplos de estructuras de datos desde su aplicación en las matemáticas (grupo, anillo, etc.) hasta en la vida real, por ejemplo, la estructura de una empresa. Los elementos de una estructura de datos podrán ser expresados según los tipos de datos que posee el lenguaje de programación en el que serán implementados.

Las estructuras de datos se caracterizarán, por lo tanto, por el tipo de elementos de la estructura, las relaciones definidas sobre los elementos y las operaciones permitidas sobre la estructura. Algunas de las operaciones más típicas efectuadas sobre estructuras de datos son: acceso o búsqueda de un elemento, inserción de un nuevo elemento, borrado de un elemento de la estructura, etc.

Se puede establecer la siguiente clasificación básica de estructuras de datos:

- Estructuras de datos **lineales**. En este tipo de estructuras cada elemento tendrá un elemento que será el 'siguiente'. Las cadenas de caracteres, arrays, listas, pilas y colas son ejemplos de este tipo de estructuras.

- Estructuras de datos **no lineales**. Un ejemplo de este tipo de estructuras serán el árbol y grafo. A este tipo de estructuras también se les denomina estructuras multienlazadas.

En este apartado se presenta la estructura de datos árbol, definiendo la estructura, algunos de sus conceptos más importantes y mostrando, por último, un ejemplo práctico de la implementación de un árbol en el lenguaje C++.

6.2. Árboles

La librería estándar de C++ incluye la definición y uso de estructuras de datos genéricas, como 'lista', 'pila' y 'cola'. Existen además otras estructuras que no están definidas en esta librería, como por ejemplo el **árbol**. En este apartado se definirá la estructura árbol y se mostrará un ejemplo completo de su implementación en el lenguaje C++.

Un árbol puede definirse como una estructura de datos no lineal y homogénea que establece una jerarquía entre sus elementos.

Esta estructura es una de las más utilizadas en informática, ya que se adapta a muchos de sus campos debido al hecho de que representa informaciones homogéneas organizadas y es una estructura que se puede manipular de una forma fácil y rápida. Algunas de las implementaciones que pueden ser realizadas con árboles son: métodos de clasificación y búsqueda algorítmica, en compilación: árboles sintácticos para representar expresiones de un lenguaje, y en inteligencia artificial: árboles de decisiones, de resolución, etc. Esta estructura árbol se utiliza para representar por lo tanto datos con una relación jerárquica entre sus elementos.

Un árbol es un conjunto de nodos, que puede ser:

- Vacío, cuando no tiene ningún elemento.
- O bien estar compuesto por un elemento raíz 'R' y n elementos llamados subárboles de R. La raíz del árbol es un nodo especial del que parten todos los subárboles descendientes. Por su parte, los nodos hoja son aquellos nodos que no tienen ningún descendiente.

La definición de un árbol implica una estructura recursiva. Es decir, la definición de un árbol se refiere a otros árboles.

Un árbol puede representarse gráficamente de la siguiente forma:

En la figura 6.1 puede observarse la jerarquía de un árbol.

A continuación se relacionan algunos términos básicos de la estructura árbol:

- En la figura puede observarse la **jerarquía** de un árbol. En ella, el nodo A es el **padre** de los nodos B y E. De la misma forma, B y E son nodos **hijo** del nodo A. El nodo A es, además, el nodo raíz del árbol.
- Un nodo **hoja** es aquel nodo que no tiene ningún subárbol.
- La **altura** o **profundidad** de un árbol es el número máximo de nodos de una rama. La altura del árbol nulo se define como -1.
- El término **grado** de un nodo se refiere al número de subárboles que tiene. Y el **grado de un árbol** será el máximo de los grados de sus nodos. Un árbol n-ario es aquel árbol en el que cada nodo es, como máximo, de

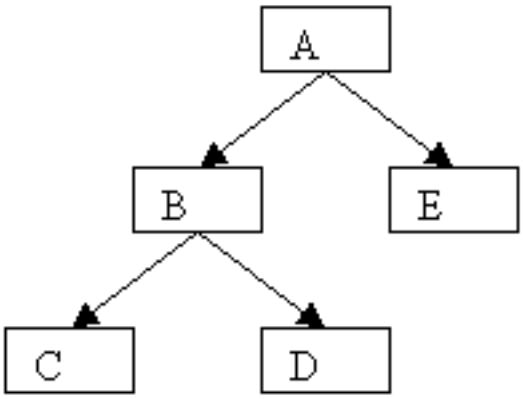


Figura 6.1: Árbol binario

grado n. Un árbol **binario** es un árbol de grado 2 (es decir, cada nodo podrá tener 0, 1 ó 2 subárboles), a cada subárbol se le llamará subárbol derecho y subárbol izquierdo. Por lo tanto, B es el hijo izquierdo de A y E es el hijo derecho de A.

6.2.1. Representación y recorrido de los árboles binarios

Los árboles binarios pueden ser representados de varias formas. Algunas de éllas serían:

- *Mediante punteros.* Cada nodo de un árbol contendrá un campo con un tipo de datos y dos punteros, apuntando a cada nodo del árbol. Esta implementación será mostrada en el siguiente apartado.
- *Mediante arrays*
- *Mediante listas enlazadas*

El recorrido de un árbol es el proceso que permite recorrer el árbol accediendo una sola vez a cada nodo del árbol. Las actividades comunes repetidas en los distintos recorridos de un árbol son las siguientes:

- *Visitar el nodo raíz.*
- *Recorrer el subárbol izquierdo.*
- *Recorrer el subárbol derecho.*

cc BY NC SA

Los recorridos de árbol más utilizados tienen en común el recorrido del subárbol izquierdo y recorrido del subárbol derecho y dependiendo de cuando se visita el nodo raíz se llaman respectivamente: **pre-orden** (primero se visita la raíz), **post-orden** (la raíz se visita la última) e **in-orden** (la raíz se visita en medio de los dos recorridos).

6.2.2. Árbol binario de búsqueda

El árbol binario de búsqueda es una variante del árbol binario que se construye teniendo en cuenta los siguientes requerimientos:

- El primer elemento se utiliza para crear el nodo raíz.
- Los valores del árbol deben ser valores que permitan establecer un orden en el árbol. Estos valores estarán identificados por una clave y no existirán dos elementos con la misma clave.
- En cualquier nodo, las claves de su subárbol izquierdo serán menores que la clave de ese nodo y las claves de su subárbol derecho serán mayores que la clave de dicho nodo.

Teniendo el árbol organizado de esta forma, al recorrer el árbol de forma in-orden, obtendremos los valores ordenados por orden.

En las búsquedas realizadas en un árbol binario de búsqueda la rama elegida en un nodo dependerá del resultado de la comparación de la clave buscada con el valor almacenado en el nodo, de forma que si la clave buscada es menor que el valor almacenado, se elegirá la rama izquierda y si es mayor, se seleccionará la rama derecha.

Los árboles binarios de búsqueda son también llamados árboles binarios ordenados. Un ejemplo de un árbol binario de búsqueda o árbol binario ordenado (ABO) es mostrado en la figura 6.2.

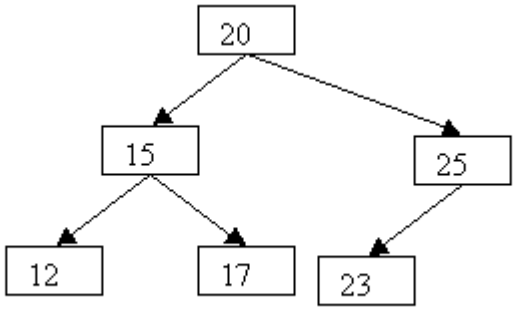


Figura 6.2: Árbol binario ordenado

6.2.3. Árboles balanceados

Como se ha comentado anteriormente, la profundidad de un árbol es el número máximo de nodos de una rama. Un árbol **balanceado** se define como un árbol que tiene todas las hojas a la misma profundidad y un árbol no balanceado es aquél que tiene ramas más largas que otras. A la hora de realizar búsquedas en un árbol es bastante útil el balanceo del árbol. Una operación de búsqueda puede visitar n nodos en un árbol no balanceado que indexa un fichero de n registros; por su parte, en un fichero igual con estructura de árbol balanceado de orden d no se visitarían más de $1+log_d(n)$ nodos. Teniendo en cuenta que cada visita requiere un acceso a dispositivo de almacenamiento secundario, el balanceo del árbol tiene un gran ahorro. A la hora de realizar el balanceo de un árbol hay que tener en cuenta que el ahorro en las operaciones de recuperación de datos sea mayor que el tiempo empleado en balancear el árbol. En la figura 6.3 se ve un esquema de árbol binario no balanceado y otro de un árbol binario balanceado.

Figura 6.3: Árbol binario no balanceado y balanceado

6.2.4. Algoritmos principales de un ABO

Las operaciones como inserción y eliminación de nodos en un árbol modifican su estructura, por este motivo, estas dos operaciones se verán en este apartado más detenidamente. Además, como se ha visto anteriormente, existen otras operaciones en el árbol, como las que realizan un recorrido del árbol que no modifican su estructura; veremos un ejemplo de este tipo de operaciones en este apartado.

6.2.4.1. Inserción de nodos

La inserción de un nodo en un ABO se realiza de la siguiente forma: debemos comparar el valor del nuevo nodo a insertar con valores almacenados en los nodos que componen el árbol. Comenzando por el nodo raíz compararemos su valor con el del nuevo nodo a insertar. Si no existe un nodo raíz, el nuevo nodo

será la raíz del nuevo árbol; por el contrario, si existe un valor almacenado en la raíz, si el nuevo nodo es mayor que su valor, continuaremos la comparación por su subárbol derecho; si es menor, continuaremos por su subárbol izquierdo y así sucesivamente hasta encontrar un nodo hoja, al cual asociaremos el nuevo nodo (enlazándolo a su rama derecha o izquierda dependiendo si el valor del nodo es mayor o menor al valor almacenado en dicho nodo hoja). Un ejemplo de inserción es mostrado en la figura 6.4.

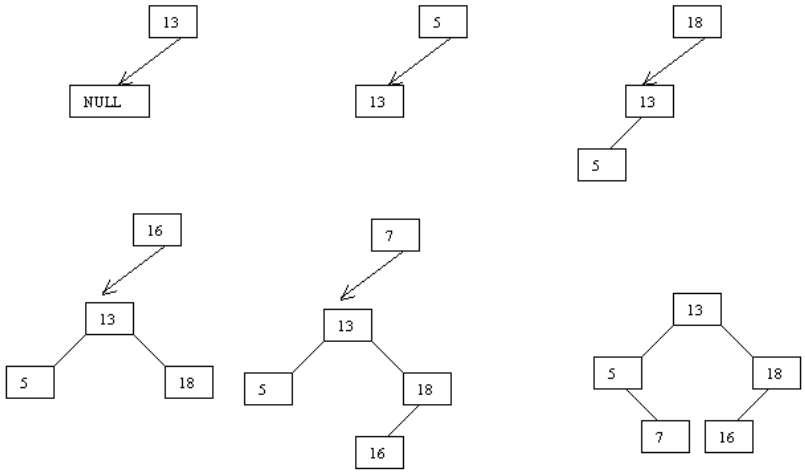


Figura 6.4: Inserción de nodos en un árbol binario ordenado

6.2.4.2. Eliminación de nodos

La eliminación de un nodo es una operación más costosa que la inserción, ya que, dependiendo del tipo de nodo a eliminar, tendremos tres casos de eliminación de nodos:

- Si el nodo a eliminar **es una hoja** (figura 6.5), lo único que habría que hacer sería eliminar la memoria ocupada por el nodo y poner a NULL el puntero del padre de ese nodo que apuntaba a él.
- Si el nodo a eliminar tiene **un hijo** (figura 6.6) tendríamos que hacer que el puntero del padre del nodo a eliminar apuntara al hijo de ese nodo y por último liberar la memoria que ocupa el nodo a eliminar.
- En el caso de que el nodo a eliminar tenga **dos hijos**, existen varias formas para solucionar esta situación, una de ellas podría realizarse sustituyendo el valor del nodo que va a ser eliminado por el valor que es su **inmediato**

predecesor; éste sería el valor almacenado en el nodo más a la derecha del subárbol izquierdo del nodo a eliminar. Deberíamos, en primer lugar encontrar el valor del inmediato predecesor; una vez encontrado, el valor del nodo a eliminar sería sustituido por el valor almacenado en ese nodo, y a continuación se liberaría la memoria que ese nodo ocupa.

Esta situación es la más complicada para eliminar nodos en un árbol; la clase de eliminación de nodos seleccionada es llamada **eliminación por copia**. Un ejemplo de su aplicación es mostrado en la figura 6.7.

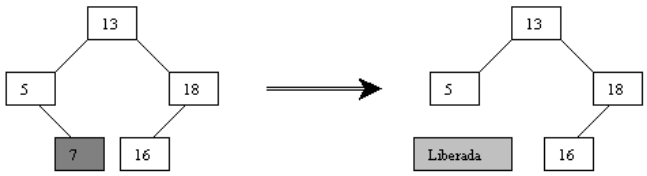


Figura 6.5: Eliminación de nodos hoja

Los algoritmos de inserción y eliminación de nodos en un árbol son mostrados en el apartado siguiente, donde se implementa un ABO completo en C++.

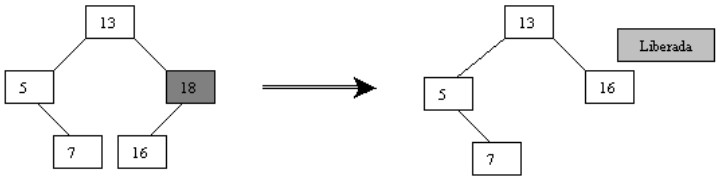


Figura 6.6: Eliminación de nodos con un hijo

6.2.4.3. Recorridos del árbol

Como se ha visto anteriormente en este capítulo, se han establecido tres formas estándar de recorrer un árbol binario, éstas son:

- **Pre-orden**: el recorrido realizado en el árbol sería en primer lugar visitar la raíz, después el subárbol izquierdo y por último el subárbol derecho.

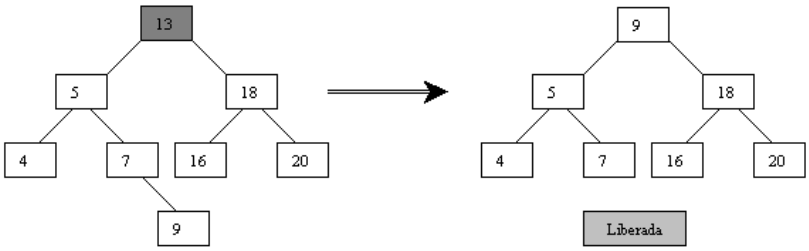


Figura 6.7: Eliminación de nodos con dos hijos

- **In-orden**: el recorrido realizado en el árbol comenzaría por el subárbol izquierdo, a continuación se visitaría la raíz y por último se recorrería el subárbol derecho.
- **Post-orden**: en este caso, en primer lugar recorreríamos el subárbol izquierdo, a continuación se recorrería el subárbol derecho y por último, se visitaría la raíz.

Un ejemplo de los tres recorridos distintos del árbol es mostrado en la figura 6.8 y la implementación del recorrido in-orden es mostrada como ejemplo en el siguiente apartado, dentro de la implementación en C++ de un ABO.

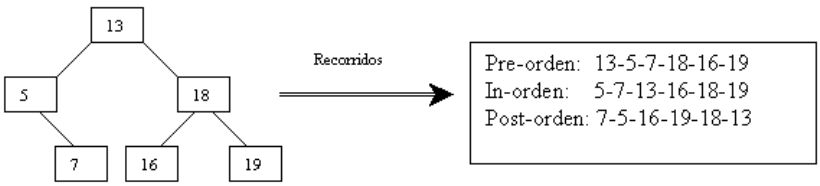


Figura 6.8: Recorridos en un árbol binario ordenado

6.2.5. Implementación de un árbol binario ordenado

En este apartado veremos la implementación en el lenguaje C++ de un árbol binario ordenado.

La implementación de un ABO en el lenguaje C++ puede ser realizada de forma dinámica, como se muestra a continuación. Se define una clase llamada

Arbol. Esta clase contendrá un elemento del tipo de datos que formen el árbol (enteros, cadenas de caracteres, etc), en el ejemplo el árbol estará compuesto de punteros a enteros. Además, consta de dos variables que son apuntadores a los subárboles izquierdo y derecho de cada nodo del árbol binario (en la implementación mostrada, estos dos apuntadores se llaman *hizq* y *hder* respectivamente).

```
typedef int *tipodato;
class Arbol {
  tipodato Raiz;
  Arbol *hizq, *hder;

   Arbol *borrarOrden(tipodato dato);
public:
  Arbol(){
    Raiz=NULL;
    hizq=hder=NULL;
  }

  Arbol(Arbol* hizq,tipodato dato,Arbol* hder) {
    this->hizq=hizq;
    this->Raiz=dato;
    this->hder=hder;
  }

  Arbol *hijoizq(){return hizq;}
  Arbol *hijoder(){return hder;}
  tipodato raiz(){return Raiz;}

  int vacio();
  void insertar(tipodato dato);
  int pertenece(tipodato dato);
  void borrar(tipodato dato);
  void inorden();
  ~Arbol();
};
int Arbol::vacio() {
  if (raiz() == NULL)
    return 1;
  else return 0;
}
void Arbol::insertar(tipodato dato) {
  Arbol *aux;
  if (vacio())
    Raiz=dato;
```

```
  else
    if (*Raiz!=*dato) {
      if (*dato<*Raiz) {
        if ((aux=hijoizq())==NULL)
          hizq=aux= new Arbol();
        }
        else {
          if ((aux=hijoder())==NULL)
            hder=aux= new Arbol();
        }
      aux->insertar(dato);
    }
    else
       cout<<"Ya existe el elemento en del arbol."<<endl;
}
int Arbol::pertenece (tipodato dato) {
  Arbol *aux;
  if (vacio())
    return 0;
  else
    if (*Raiz!=*dato) {
      if (*dato<*Raiz)
        aux=hijoizq();
      else
        redundante aux=hijoder();
      if (aux==NULL)
        return 0;
      return aux->pertenece(dato);
    }
   else
      return 1;
}
void Arbol::borrar(tipodato dato) {
  if (!vacio()) {
    if (*dato<*Raiz) {
      if (hizq != NULL)
        hizq = hizq->borrarOrden(dato);
    }
  else
      if (*dato>*Raiz) {
        if (hder != NULL) {
          hder = hder->borrarOrden(dato);
        }
     else //En este caso el dato es la raiz {
```

```
        if (hizq==NULL && hder==NULL) {
         delete Raiz;
         Raiz=NULL;
        }
       else
         borrarOrden(dato);
      }
    }
  }
  Arbol *Arbol::borrarOrden(tipodato dato) {
     tipodato datoaux;
     Arbol *retorno=this, *aborrar, *candidato, *antecesor;
    if (!vacio()) {
      if (*dato<*Raiz) {
        if (hizq != NULL)
          hizq = hizq->borrarOrden(dato);
      }
      else
        if (*dato>*Raiz){
          if (hder != NULL) {
            hder = hder->borrarOrden(dato);
          }
        else {
          aborrar=this;
          if ((hDer==NULL)&&(hIzq==NULL))
                  { /*si es hoja*/
                      delete aborrar;
                      retorno=NULL;
                  }
                  else
                  {
                      if (hDer==NULL)
                      { /*Solo hijo izquierdo*/
                          aborrar=hIzq;
                          datoaux=datoRaiz;
                          datoRaiz=hIzq->raiz();
                          hIzq->datoRaiz = datoaux;
                          hIzq=hIzq->hijoIzq();
                          hDer=aborrar->hijoDer();
                          retorno=this;
                      }
                      else
                          if (hIzq==NULL)
                          { /*Solo hijo derecho*/
```

```
                                aborrar=hDer;
                                datoaux=datoRaiz;
                                datoRaiz=hDer->raiz();
                                hDer->datoRaiz = datoaux;
                                hDer=hDer->hijoDer();
                                hIzq=aborrar->hijoIzq();
                                retorno=this;
                            }
                            else
                            { /* Tiene dos hijos */
                                candidato = hijoIzq();
                                antecesor = this;
                                while (candidato->hijoDer())
                                {
                                    antecesor = candidato;
                                    candidato = candidato->hijoDer();
                                }
                                /*Intercambio de datos de candidato*/
                                datoaux = datoRaiz;
                                datoRaiz = candidato->raiz();
                                candidato->datoRaiz=datoaux;
                                aborrar = candidato;
                                if (antecesor==this)
                                    hIzq=candidato->hijoIzq();
                                else
                                    antecesor->hDer=candidato->hijoIzq();
                            } //Eliminar solo ese nodo, no todo el subarbol
                        aborrar->hIzq=NULL;
                        aborrar->hDer=NULL;
                        delete aborrar;
                    }
                }
        }
    return retorno;
}
void Arbol::inorden() {
    Arbol *aux;
    if (!vacio()) {
        if (aux = hijoizq())
            aux->inorden();
        cout<<*(Raiz)<<endl;
        if (aux = hijoder())
            aux->inorden();
    }
```

```
}
Arbol::~Arbol() {
  Arbol *aux;
  if (!vacio()) {
    if (aux=hijoizq())
      delete aux;
    if (aux=hijoder())
      delete aux;
    if (Raiz)
      delete Raiz;
  }
}
void main(void) {
  Arbol miarbol;
  int *pint,j;
  //insertamos varios valores en el árbol
  pint = new int;
  *pint = 5;
  miarbol.insertar(pint);
  pint = new int;
  *pint = 3;
  miarbol.insertar(pint);
  pint = new int;
  *pint = 1;
  miarbol.insertar(pint);
  pint = new int;
  *pint = 9;
  miarbol.insertar(pint);
  pint = new int;
  *pint = 7;
  miarbol.insertar(pint);
  pint = new int;
  *pint=9;
  cout<<"Buscando un nodo"<<endl;
  if (miarbol.pertenece(pint)==1)
    cout<<"Encontrado"<<endl;
  delete pint;
  cout<<"Recorriendo en inorden el árbol"<<endl;
  miarbol.inorden();
}
```

En la implementación del árbol cabe destacar lo siguiente:

- Se ha definido el tipo de datos *tipodato* para hacer más fácil la comprensión al lector del ejemplo desarrollado. En este ejemplo se ha imple-

mentado un árbol que está formado por nodos que contienen punteros a enteros y dos apuntadores a sus respectivos subárboles. Cada nodo está compuesto de un entero, llamado *Raiz* en el ejemplo, y dos apuntadores, apuntando cada uno al subárbol izquierdo y derecho del nodo, llamados *hizq* y *hder,* respectivamente.

Como se ha comentado, en este ejemplo se ha utilizado la estructura árbol para almacenar enteros, pero podría haberse implementado un árbol que estuviera formado por datos que pudieran mantener un orden como **float** o **double**; o datos para los que los operadores de comparación estuvieran sobrecargados para poder realizar inserciones y comparaciones en el árbol (un ejemplo podrían ser las cadenas de caracteres).

- Se han definido las siguientes funciones miembro dentro de la clase:
  - *Dos constructores*, uno que inicializa a NULL tanto el puntero a enteros que define el nodo como los apuntadores *hizq* y *hder*; y un segundo constructor que utiliza como parámetro de entrada un objeto de tipo árbol.
  - *Vacio*: esta función dice si el árbol está vacío o no. Es muy útil a la hora de insertar nuevos nodos en el árbol.
  - *Insertar*. Función miembro que inserta un nodo en el árbol. Recibe como parámetro de entrada el puntero a entero que debe insertarse en el árbol.
  - *Pertenece*. Función miembro que busca por el árbol un valor introducido por parámetro y devuelve un valor 1 si existe o en caso contrario devuelve un valor 0. Recibe como parámetro de entrada el puntero a entero que debe ser buscado en el árbol.
  - *Borrar*. Elimina un nodo determinado del árbol. Recibe como parámetro de entrada el puntero a entero que debe eliminarse del árbol. Esta función utiliza a su vez la función privada borrarOrden.
  - *Inorden*. Esta función es utilizada para mostrar el árbol completo, recorriendo el árbol siguiendo el orden siguiente en cada nodo: subárbol izquierdo, la raíz y el subárbol derecho.
  - *Hijoizq*. Esta función devuelve el subárbol izquierdo de un nodo.
  - *Hijoder*. Devuelve el subárbol derecho de un nodo.
  - *Raiz*. Devuelve el puntero a entero almacenado en un nodo determinado.
  - *Destructor*. Esta función elimina uno a uno cada nodo del árbol.

La mayoría de estas funciones trabajan de forma recursiva. Además son totalmente orientadas a objeto, de forma que cada nodo del árbol es tratado como un objeto *Arbol* por sí mismo.

6.3. Resumen

En este capítulo se ha prensentado la estructura de datos **árbol**. Ésta es una de las estructuras de datos no lineales más utilizadas en informática. Se han definido los conceptos más importantes que forman la estructura, así como distintas formas de recorrido y representación de árboles binarios ordenados. Para finalizar, se ha realizado una implementación práctica de un árbol binario ordenado en C++.

cc BY NC SA

Capítulo 7

Identificación y modelado

Durante todo el libro hemos hablado de clases, de objetos, de herencia, de composición, etc. En este capítulo nos vamos a centrar en cómo, dado un problema real, saber reconocer qué objetos lo componen, sus atributos, las operaciones disponibles sobre esos atributos, cómo se relacionan e interactúan los objetos entre ellos y cómo representar el sistema obtenido, mediante modelado estático, que ayuda a representar la arquitectura de un sistema, y dinámico, que expresa el comportamiento de las clases de un sistema, de manera formal. En cuanto a este último objetivo, vamos a hacer, a lo largo del capítulo, una introducción a la notación más utilizada para el modelado de sistemas software: UML (Lenguaje Unificado de Modelado). Concretamente, nos centraremos en tres tipos de diagrama dentro de este lenguaje que consideramos útiles para los lectores: el diagrama de casos de uso, el diagrama de clases y el diagrama de secuencias.

Para ilustrar cada fase de identificación, el capítulo se encuentra dividido en cinco partes relacionadas a través de un problema que se va resolviendo en cada apartado. En la primera parte, el objetivo es aprender a identificar las clases de objetos a partir de un enunciado dado. En la segunda parte, se persigue identificar los atributos de las clases obtenidas. La tercera tiene como fin identificar las operaciones de cada una de las clases y se explica el diagrama de casos de uso para ayudar en esta labor. En la cuarta parte, se expone qué es un diagrama de clases como forma más extendida de representar gráficamente el modelado estático de clases, llevando a este diagrama todo lo obtenido en los tres apartados anteriores. Por último, en la quinta parte, se presenta el diagrama de secuencias como forma de representar el modelado dinámico y se aplica a las operaciones más representativas de nuestro problema.

A continuación se expone el problema que vamos a utilizar durante todo el capítulo.

Una empresa de alquiler de vehículos nos ha encargado la construcción de un sistema software. Con este fin nos ha facilitado la siguiente descripción del

sistema.

La empresa posee una amplia flota de vehículos divididos en dos tipos: turismos y comerciales. Además de los datos básicos que ambos tipos tienen en común (nº bastidor, matrícula, marca, modelo y número de vehículo de la empresa), debe almacenarse el número de plazas de los primeros y el peso soportado de los segundos.

Cuando un cliente solicita un vehículo debe indicar la fecha de recogida, la fecha de devolución, el tipo, junto con el número de plazas si es turismo, o el peso soportado si es un vehículo comercial, y el número de tarjeta de crédito con que se abonará el alquiler.

La empresa comprueba la disponibilidad en ese momento de algún vehículo del tipo solicitado: si lo hay, se lo alquila al cliente, debiendo almacenar el sistema la fecha de recogida, la fecha de devolución, el número del vehículo alquilado, el importe del alquiler, calculado a partir de precio por día del tipo de vehículo y los días que va a estar alquilado, y el número de tarjeta de crédito del cliente. En caso de no tener un vehículo disponible, se lo comunica al cliente y se anula la solicitud.

La empresa desea que el sistema, a partir de una fecha que se le indique, devuelva un listado con todos los importes de los alquileres y los números de tarjeta de crédito.

7.1. Identificación

En realidad, cuando se habla de identificar objetos, realmente se habla de identificar clases de objetos. A lo largo de este capítulo utilizaremos indistintamente **clase de objeto** u **objeto** para referirnos a la estructura que encapsula datos y operaciones, e **instancia** de objeto a una ocurrencia de esa clase.

7.1.1. Análisis gramatical para búsqueda de clases

Una de las propuestas para identificar clases de objetos fue realizada por Abbott en 1983. Esta propuesta recomienda que, dada una descripción del sistema en lenguaje natural, realicemos un análisis gramatical, donde busquemos verbos y nombres, que serán candidatos a ser operaciones o relaciones, y clases o atributos, respectivamente. Esta propuesta es útil para modelar problemas sencillos, aunque hay que tener cuidado para que el estilo de escritura del autor del documento no influya en el análisis realizado, es decir, hay que intentar leer entre líneas, ideas o conceptos, que pueden estar implícitos en el documento.

En el caso de problemas complejos es obligatorio recurrir a cualquier tipo de documentación relacionada con el sistema que pueda aportarnos nuevos elementos al modelado. Tampoco hay que olvidar a los futuros usuarios, que seguramente podrán aportarnos más conocimiento acerca del sistema a modelar. Las dos últimas apreciaciones quizás pertenecen más al ámbito de un libro

de ingeniería del software que a uno de programación, pero creemos necesario que el lector no olvide que antes de ponerse a programar hay que estudiar con detenimiento qué queremos construir y con qué objetivo.

Una vez que ya hemos obtenido todos los términos relevantes, hay que evaluar la conveniencia o no de cada término obtenido, utilizando una de las muchas propuestas que existen en la literatura sobre identificación de clases.

Siguiendo esta propuesta, lo primero que debemos hacer es una lista con los nombres significativos que aparecen en el enunciado. Esta lista de nombres aparece en el cuadro 7.1.

Nombres significativos / Candidatos a clase
Empresa de alquiler de vehículos
Vehículos
Tipo de vehículo
Turismos
Comerciales
Nº bastidor
Matrícula
Marca
Modelo
Nº de vehículo de la empresa
Plazas
Peso
Cliente
Fecha de Recogida
Fecha de Devolución
Número de tarjeta de crédito
Alquiler
Disponibilidad
Importe
Precio por día
Nº Días
Fecha
Listado

Cuadro 7.1: Nombres del enunciado

Hay que tener en cuenta que es muy posible que los candidatos a objetos rechazados sean posibles atributos de los objetos que finalmente se acepten.

7.1.2. Clasificación de clases potenciales

Una vez que hemos localizado todos los nombres en el enunciado, podemos clasificarlos en función de las diversas formas en que los objetos pueden aparecer, existen diversas propuestas de clasificación de objetos, nosotros usaremos la siguiente:

- **Entidades externas**, que proporcionan o reciben información del sistema. En nuestro ejemplo no hay, pero por ejemplo, si el sistema interactuara con un banco, éste sería una entidad externa.
- **Elementos**, que forman parte del dominio de información del problema. Por ejemplo, el número de tarjeta de crédito.
- **Sucesos**, que transcurren durante una operación del sistema. Por ejemplo, el alquiler de un vehículo.
- **Roles**, que realizan personas que interactúan con el sistema. Por ejemplo, cliente.
- **Unidades organizativas**. Por ejemplo Turismos y Comerciales.
- **Lugares**, que sitúan el contexto del problema. Por ejemplo, la empresa de alquiler entendido como espacio físico donde se realizan los alquileres.
- **Estructuras**, que especifican una clase de objeto. Por ejemplo, vehículos.

En función de esta clasificación, podemos aplicarla sobre nuestra lista de nombres como se ve en el cuadro 7.2.

Como hemos visto, en esta primera clasificación ya hemos empezado a distinguir atributos de candidatos a clases de objetos.

7.1.3. Clases obtenidas

Existe una propuesta realizada por Coad y Yourdon, en 1991, que enuncia 6 características que un candidato a clase debe intentar cumplir en su totalidad (o casi). Estas características de selección son las siguientes:

1. **Información retenida**: un objeto es útil si la información sobre él es necesaria para que el sistema funcione.
2. **Servicios necesarios**: el objeto debe contener una serie de operaciones identificables que puedan cambiar el valor de sus atributos.
3. **Atributos múltiples**: objetos con un solo atributo quizás sea mejor intentar ubicarlos como atributos de otro objeto.
4. **Atributos comunes**: los atributos de una clase de objeto son válidos para todas las posibles instancias de ese objeto.

Nombres significativos / Candidatos a clase	**Clasificación**
Empresa de alquiler de vehículos	Lugar
Vehículos	Estructura
Tipo de vehículo	No objeto. Posible atributo de vehículo
Turismos	Unidad organizativa
Comerciales	Unidad organizativa
Nº bastidor	No objeto. Posible atributo de vehículo
Matrícula	No objeto. Posible atributo de vehículo
Marca	No objeto. Posible atributo de vehículo
Modelo	No objeto. Posible atributo de vehículo
Nº de vehículo de la empresa	No objeto. Posible atributo de vehículo
Plazas	No objeto. Posible atributo de vehículo
Peso	No objeto. Posible atributo de vehículo
Cliente	Entidad externa o rol
Fecha de Recogida	No objeto. Posible atributo de alquiler de vehículo
Fecha de Devolución	No objeto. Posible atributo de alquiler de vehículo
Número de tarjeta de crédito	No objeto. Posible atributo de alquiler de vehículo
Alquiler	Suceso
Disponibilidad	No objeto. Posible resultado operación sobre vehículo
Importe	No objeto. Posible atributo de alquiler de vehículo
Precio por día	No objeto. Posible atributo de vehículo
Nº Días	No objeto. Posible resultado operación
Fecha	No objeto. Posible atributo de alquiler de vehículo
Listado	No objeto. Posible resultado operación

Cuadro 7.2: Clasificación de nombres del enunciado

cc BY NC SA

5. **Operaciones comunes**: las operaciones de una clase de objeto son válidas para todas las posibles instancias de ese objeto.

6. **Requisitos esenciales**: entidades externas que proporcionan o reciben información imprescindible para el sistema.

Aplicando estas seis características sobre los posibles candidatos del cuadro 7.2 obtenemos el cuadro 7.3 con las clases aceptadas y rechazadas.

Nombres significativos / Candidatos a clase	**Clasificación**
Empresa de alquiler de vehículos	Aceptado
Vehículos	Aceptado
Turismos	Aceptado
Comerciales	Aceptado
Cliente	Rechazado
Alquiler	Aceptado

Cuadro 7.3: Clasificación de nombres del enunciado

Después de hacer esta evaluación, nuestro sistema va a tener los objetos e*mpresa de alquiler de vehículos*, *vehículos*, *turismos*, *comerciales*, y *alquiler*. En la clase *empresa de alquiler de vehículos* se gestionará la flota de vehículos pertenecientes a la empresa. La clase *vehículos* deberá mantener la información y comportamiento común de los vehículos que la empresa alquila. Las clases *turismo* y *comerciales* son especializaciones de la clase *vehículos* donde mantendremos la información y comportamiento específico de esa clase de vehículos. Por último, la clase *alquiler* almacenará la información de cada vehículo alquilado por la empresa.

Se ha rechazado *Cliente* como clase porque la función del cliente es interactuar con el sistema solicitando un alquiler.

7.2. Identificación de atributos

Los atributos deben ayudarnos a describir las clases aceptadas. No en vano, son los que dan sentido al objeto dentro del contexto del problema. Esto último es muy importante, ya que el contexto del problema puede hacer variar los atributos de un objeto. Por ejemplo, si imaginamos un sistema de gestión de una librería donde se está modelando la clase *libro*, los atributos de esa clase serán diferentes al de un contexto de modelado de *libro* que sea la gestión de préstamos de una biblioteca. Así, el objeto *libro* de librería debería contener atributos como *precio de distribuidor*, *precio sin IVA* o *precio con IVA*, mientras que esos atributos no serían necesarios en la gestión de prestamos de biblioteca, donde en cambio podríamos tener atributos como la *signatura* y su *disponibilidad*.

Clases	**Atributos**
Empresa de alquiler de vehículos	Nombre Domicilio Teléfono
Vehículos	Nº Bastidor Matrícula Marca Modelo Nº Vehículo de Empresa Precio/Día Tipo
Turismos	Nº Plazas
Comerciales	Peso
Alquiler	Código alquiler Fecha alquiler Nº Vehículo de Empresa Fecha de recogida Fecha de devolución Importe Número de tarjeta de crédito

Cuadro 7.4: Clases y atributos

Es conveniente estudiar el enunciado del problema que se va a modelar para elegir los atributos necesarios que van a describir correctamente el objeto y que lo hacen único en el contexto dado. Tomemos como ejemplo la clase *vehículos* de nuestro problema. En el contexto del alquiler de coches puede ser interesante almacenar el *número de bastidor*, la *marca* y la *matrícula* para identificarlo, el *precio por día* de alquiler y las *fechas* en las que está alquilado para comprobar su disponibilidad. Atributos como el número de *plazas* o su *peso* no los vamos a especificar en *vehículos* ya que no son necesarios en el contexto de nuestro problema para todas las instancias de *vehículo*, si no que cada uno de ellos será un atributo distintivo de cada una de sus dos clases hijas: *plazas*, para *turismos*, y *peso*, para *comerciales*. Atributos que a todos nos vienen a la mente como el *color* o la *cilindrada* no los incluimos por no ser imprescindibles para el buen funcionamiento del sistema.

Además, como se indicó en el punto anterior, es bastante probable que los candidatos a objetos rechazados puedan ser atributos de los objetos aceptados.

En el cuadro 7.4 se especifica cada clase y los atributos identificados dentro de contexto de nuestro problema:

7.3. Identificación de operaciones

Las operaciones especifican el comportamiento de un objeto mediante la modificación de uno o más de los atributos del objeto. Por lo tanto, las operaciones deben conocer los atributos de un objeto y deben poder manipularlos. Las operaciones sobre objetos se pueden dividir en tres grupos:

- **Manejo de datos**: seleccionan, modifican, añaden o eliminan datos.
- **Realizar cálculos**: toman datos externos y/o del objeto y devuelven el resultado de operaciones sobre ellos.
- **Monitorización**: controlan el estado de un objeto durante algún suceso.

7.3.1. Casos de Uso y Diagramas de Casos de Uso

Es recomendable en las primeras fases de modelado de un sistema intentar comprender las relaciones entre éste y su entorno, intentando utilizar una aproximación lo más abstracta posible. Para ello, en UML, se propone una representación de cada interacción del sistema con el entorno, denominada modelo de **caso de uso**. Los casos de uso se utilizan para abstraer el comportamiento deseado del sistema a modelar, sin necesidad de entrar en detalles de cómo se implementará ese comportamiento. Desde el punto de vista de un **actor** o **entidad externa**, el caso de uso en el que esté involucrado debe reportarle algo tangible, ya sea el cambio de un objeto o la generación de uno nuevo, o la devolución del resultado de alguna operación. Es decir, el modelo de casos de uso nos sirve como punto de partida para identificar las operaciones que ofrecerá el sistema para interactuar con él. Los casos de uso se pueden aplicar tanto a sistemas como a subsistemas.

Gráficamente, los **casos de uso** se suelen representar con una elipse y su nombre dentro, el nombre debe ser diferente para cada caso de uso. Normalmente, al ser descripciones de comportamiento, los nombres de los casos de uso suelen ser expresiones verbales. En la figura 7.1. se muestran ejemplos de casos de uso.

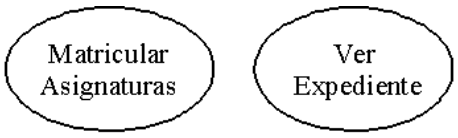


Figura 7.1: Casos de uso

Un **actor** participante es la persona, sistema existente, u objeto involucrado en la acción. Por ejemplo, en un sistema de gestión de una universidad, actores pueden ser *alumno* o *secretaría*. En este modelo, la entidad externa o actor

participante es representada como una especie de figura humana lineal. En la figura 7.2 se muestran ejemplos de actores.

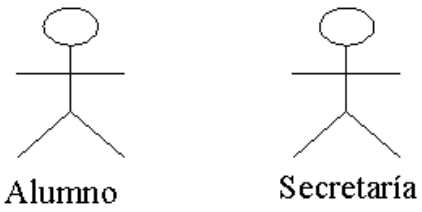


Figura 7.2: Actores

Normalmente, los casos de uso, se suelen utilizar para modelar el **contexto de un sistema**, lo que implica enmarcar todo el sistema y especificar los actores que interactúan con él. Supongamos, como ejemplo, que tenemos que modelar el sistema de gestión de expedientes de una *universidad*. Dentro del sistema tendremos elementos como *asignaturas* y *matriculaciones*, que serán los encargados de realizar el comportamiento esperado por los elementos externos. Fuera del sistema estarán los elementos externos que interactúan con el sistema y que pueden ser *alumnos* y *secretaría*. Al especificar qué actores se incluyen y cuales no, estamos restringiendo el entorno del sistema.

La figura 7.3 muestra el contexto de un sistema de gestión de expedientes, y dentro de este, un subsistema, bastante simple, donde sólo nos vamos a centrar en mantener el histórico de asignaturas en las que se matrícula el *alumno*, que es, junto con *secretaría*, uno de los actores. En este ejemplo vemos que un actor, *alumno*, está iniciando un caso de uso, *matricular asignaturas*, y como ese mismo caso de uso le da información al otro actor, *secretaría*, para *gestionar expediente*. Además el *alumno* puede *ver la información* de su expediente.

7.3.1.1. Diagrama de Casos de Uso de nuestro sistema

Lo primero que debemos analizar es quién interactuará con nuestro sistema, es decir, los actores del sistema. Es fácil ver que, el primer actor es el *Cliente*, que quiere alquilar un *vehículo* y que quedó descartado como clase del sistema. El segundo actor que interactúa con el sistema no aparece explícitamente pero puede extraerse leyendo el enunciado, ya que nos indica que el sistema debe devolver un listado de alquileres. Este listado de alquileres es de suponer que será solicitado por algún *empleado* de la empresa.

El segundo paso debe ser identificar las operaciones que ofrecerá el sistema a los actores encontrados. En el caso del *Cliente*, se le ofrece la operación *Solicitar Alquiler*. En el caso de empleados de la empresa también hemos visto que una operación que podrá realizar es la petición de un *listado* de alquileres. Además de estos dos casos de uso, y leyendo detenidamente el enunciado, vemos

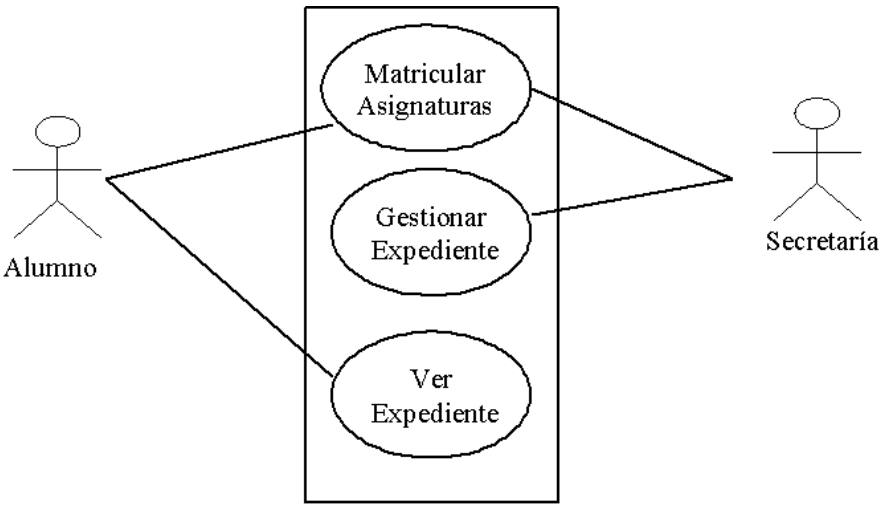


Figura 7.3: Diagrama de Casos de Uso de un sistema

que la empresa posee una flota de vehículos. El *alta* y *baja* de vehículos en dicha flota son posibles operaciones que debería realizar nuestro sistema, y que, normalmente, deberá poder realizarlas un empleado de la empresa.

Por tanto, en la figura 7.4. puede verse plasmado el análisis de casos de uso de nuestro sistema en el diagrama correspondiente.

7.3.2. Análisis gramatical de verbos

El siguiente paso para obtener las operaciones de los objetos es volver a estudiar detenidamente el problema, y realizar un nuevo análisis gramatical, pero aislando en esta ocasión los verbos significativos. Así, el cuadro 7.5 muestra el resultado de este análisis sobre nuestro enunciado.

Verbos / Candidatos a operaciones
Posee (vehículos)
Solicita (un vehículo)
Abonará (el alquiler)
Comprueba (disponibilidad)
Alquila (vehículo)
Anula (el alquiler)
Devuelve (listado)

Cuadro 7.5: Verbos del enunciado

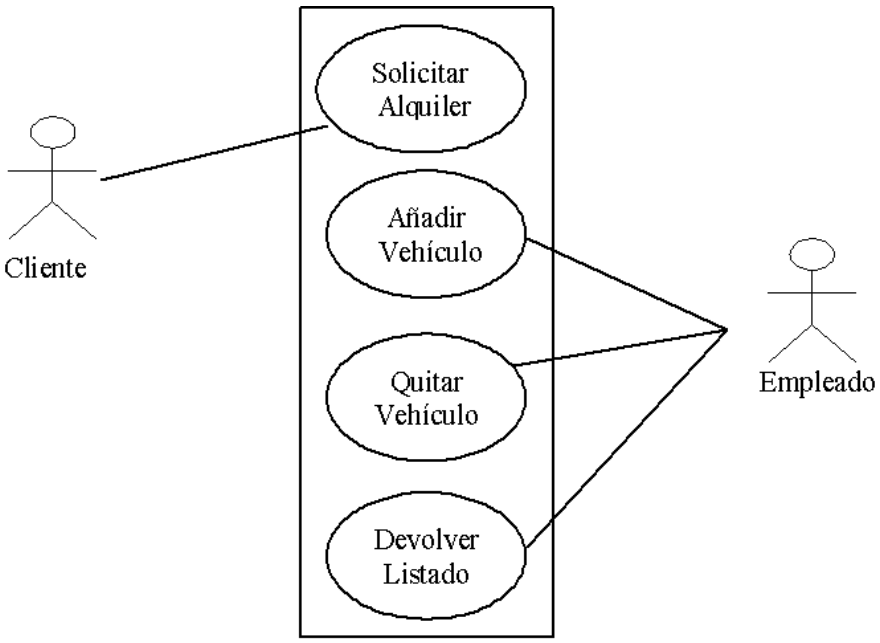


Figura 7.4: Diagrama de Casos de Uso de nuestro sistema

7.3.3. Estudio de las acciones

El siguiente paso debe ser analizar si los verbos del enunciado serán operaciones de nuestros objetos o no. Para llevar a cabo este análisis es conveniente comprobar si el verbo realiza alguno de los tres tipos de operaciones descritos anteriormente sobre un objeto. Esta comprobación se ve en el cuadro 7.6.

Normalmente, todas las operaciones de los objetos no se encuentran explícitamente en el enunciado. Debido a esto, todavía nos queda estudiar qué operaciones son necesarias en función de los atributos que hemos asignado a los objetos, como por ejemplo, si es necesario leer el valor de un atributo para alguna de las operaciones obtenidas anteriormente.

7.3.4. Operaciones obtenidas

A partir de los casos de uso y del análisis gramatical de las acciones de nuestro sistema, podemos extraer un conjunto de operaciones para las clases identificadas. En el cuadro 7.7 y en el 7.8. se muestran estas operaciones, junto con su tipo y una breve descripción.

Verbos/Candidatos a operaciones	**Análisis**
Posee (vehículos)	No es operación, pero sugiere una relación directa entre la clase Empresa de Vehículos de Alquiler y Vehículo. La Empresa, para gestionar los vehículos que posee, deberá tener operaciones para añadir y borrar vehículos
Solicita (un alquiler)	Es una operación básica ofrecida por la clase Empresa de Alquiler de Vehículos. Se debe encargar de buscar, entre los vehículos que posee la empresa, uno del tipo y las características requeridas (peso o plazas) y que esté disponible en las fechas solicitadas. Una vez encontrado, deberá calcular el importe del alquiler y realizarlo. Quizás sea interesante dividir parte de su funcionalidad en otras operaciones a las que llame
Abonará (el alquiler)	No es una operación del sistema, porque no se ha especificado que el sistema sea responsable del cobro. Sólo atañe al sistema la devolución del listado de alquileres
Comprueba (disponibilidad)	Es una operación privada de la clase Empresa de Alquiler de Vehículos. Esta operación será llamada por Solicitar Vehículos y necesitará además una operación en la clase Alquiler que le comunique si para un vehículo pasado por parámetro y unas fechas, no se produce solapamiento con los alquileres existentes
Alquila (vehículo)	Es una operación privada de la clase Empresa de Alquiler de Vehículos que deberá realizarse cuando la solicitud de un vehículo pueda llevarse a cabo
Anula (el alquiler)	No es operación, ya que puede ir implícito como respuesta negativa a la solicitud de un alquiler
Devuelve (Listado)	Será una operación que ofrezca la clase Empresa de Alquiler de Vehículos con los vehículos alquilados desde una fecha, con el importe y la tarjeta de crédito donde realizar el cargo

Cuadro 7.6: Estudio de verbos como operaciones

Clase	Operación	Tipo	Descripción
Empresa de alquiler de vehículos	Buscar Vehículo	Privada	Busca un vehículo del tipo y característica (nº de plazas en caso de turismos o peso en caso de comerciales) solicitado por el cliente hasta encontrar uno que además esté disponible en las fechas solicitadas
	Comprobar Disponibilidad	Privada	Comprueba todos los alquileres para asegurar que un vehículo dado no está ocupado en las fechas solicitadas
	Calcular Importe	Privada	Calcula el importe de un alquiler a partir de los días de duración del mismo y del precio por día del vehículo alquilado
	Crear Alquiler	Privada	Crea una instancia de alquiler si se ha encontrado un vehículo con el tipo y características solicitadas y que esté disponible en las fechas indicadas
	Añadir Vehículo	Pública	Añade un vehículo a la flota de vehículos de la empresa
	Quitar Vehículo	Pública	Elimina un vehículo de la flota de vehículo de la empresa.
	Solicitar Alquiler	Pública	Busca un vehículo del tipo y característica solicitado por el cliente en las fechas que éste último le indique. En caso de encontrarlo, el vehículo sería alquilado
	Devolver Listado	Pública	Devuelve el listado de alquileres desde la fecha que se le indique
Vehículos	Obtener Tipo	Pública	Devuelve el tipo (comercial o turismo) de un vehículo
Vehículos	Obtener Característica	Pública	Operación abstracta que, en las clases hijas, devuelve la característica del vehículo (nº de plazas en caso de turismos o peso en caso de comerciales)
Vehículos	Obtener Precio/Día	Pública	Devuelve el precio/día de un vehículo
Vehículos	Obtener Nº Vehículo	Pública	Devuelve el nº de vehículo en la empresa de un vehículo
Turismos	Obtener Característica	Pública	Operación heredada de la clase padre vehículos que devuelve el nº de plazas de un vehículo
Comerciales	Obtener Característica	Pública	Operación heredada de la clase padre vehículos que devuelve el peso de un vehículo

Cuadro 7.7: Clases y operaciones

Clase	Operación	Tipo	Descripción
Alquiler	Comprobar Solapamiento	Pública	Devuelve si las fechas durante las que una instancia de alquiler de un vehículo coinciden con las fechas que se le indiquen
Alquiler	Obtener Datos Alquiler	Pública	Devuelve los datos de una instancia alquiler

Cuadro 7.8: Clases y operaciones

7.4. Modelado del Diseño Estático

Para expresar el modelado obtenido, existe una herramienta, denominada **diagrama de clases**, que representa gráficamente los aspectos estáticos de un sistema, recogiendo en él las clases y sus asociaciones. Es decir, este diagrama muestra las clases del sistema, su estructura y comportamiento, y las asociaciones o relaciones entre las clases, pero no nos facilita ningún tipo de información temporal. Los componentes básicos del diagrama son las **clases** y las **relaciones**.

En este apartado, primero, explicaremos el concepto de clase en este diagrama y su notación, y a continuación, haremos lo mismo con las relaciones. Para concluir, expresaremos la solución al problema propuesto en este capítulo en forma de diagrama de clases.

7.4.1. Clases

El diagrama permite representar **clases** y **clases abstractas**. En el diagrama, una clase es un conjunto de objetos con propiedades similares (atributos) y un comportamiento común (operaciones). Una clase abstracta es una clase útil conceptualmente en el modelado; pero es una clase en la que sólo se pueden instanciar sus descendientes, es decir, es una clase que no puede existir en la realidad. Suele actuar como una especie de repositorio donde las subclases de nivel inferior tienen a su disposición métodos y atributos para compartir o heredar.

En el diagrama, una clase se representa como un rectángulo dividido en tres partes por dos líneas horizontales, estas tres partes contienen:

- El **nombre** de la clase debe aparecer centrado en la parte superior y si es una clase abstracta debe aparecer en cursiva.
- La lista de **atributos** aparece en la zona intermedia, debiendo aparecer uno por línea. En función del grado de detalle al que se quiera llegar, podemos indicar sólo la visibilidad y el nombre, o también el tipo y su

valor por defecto; el valor por defecto también puede ser el devuelto por una función. La visibilidad se indica con + para pública, - para privada y # para protegida. El formato de la lista de atributos es:

```
visibilidad nombre: tipo = valor_por_defecto
```

Por ejemplo, un atributo privado que exprese la *fecha de nacimiento* de un *empleado* podría representarse como:

```
-fecha de nacimiento : Fecha
```

Imaginemos ahora que un e*mpleado* posee un atributo protegido que sea la *fecha de alta* en la empresa que por defecto queremos que sea la *fecha del sistema* en el momento de su creación, su representación sería:

```
# fecha de alta: Fecha = fecha_sistema
```

- La lista de **operaciones** que proporciona la clase se muestra en la parte inferior del rectángulo. Al igual que en el caso de los atributos puede expresarse con mayor o menor grado de detalle. El formato del mayor grado de detalle es:

```
visibilidad nombre (lista_de_parámetros): tipo_devuelto
```

La visibilidad se indica igual que en el caso de los atributos con + para pública, - para privada y # para protegida. Lista_de_parámetros es una lista que contiene los parámetros que recibe la operación separados por comas. Siendo el formato de un parámetro:

```
dirección nombre: tipo = valor_por_defecto
```

La dirección indica si es un parámetro de entrada (in, por defecto), de salida (out) o de entrada/salida (inout). Por ejemplo, supongamos que la clase *empleados* tiene una operación pública que, dado un *nombre* y dos *apellidos*, nos devuelve la *fecha de alta* en la empresa, esta operación podríamos representarla así:

```
+ Obtener_Alta (nombre: string, apellido1: string,
                apellido2: string): Fecha
```

Si no queremos tanto grado de detalle simplemente podríamos representarlo como:

```
+ Obtener_Alta ()
```

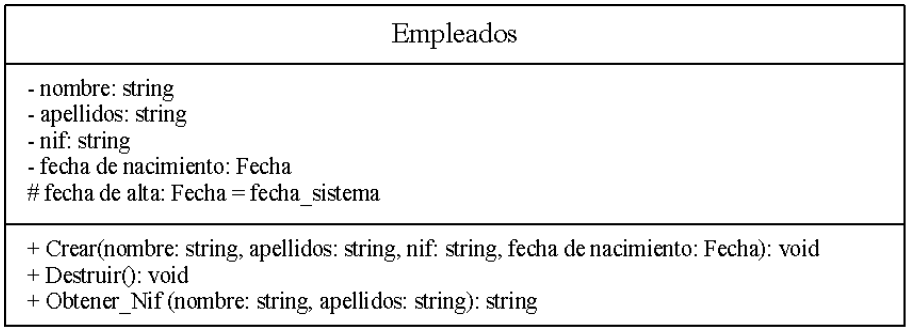


Figura 7.5: Clase empleados detallada

La representación detallada de una clase empleados podría quedar como se ve en la figura 7.5.

Aunque también sería correcta una representación tan simple como la de la figura 7.6.

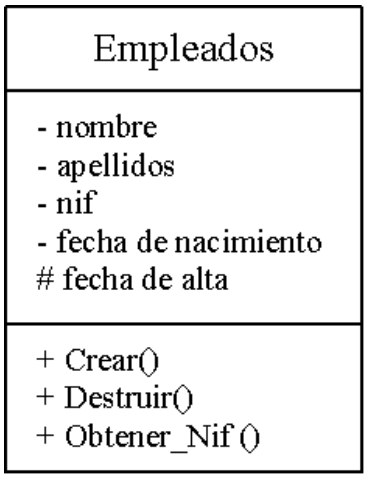


Figura 7.6: Clase empleados con menor detalle

7.4.2. Relaciones

Existen varias relaciones estáticas entre clases que se pueden modelar, nosotros nos vamos a centrar en las que consideramos más importantes: la **herencia** y la **asociación**, y dentro de esta última, la **composición** y la **clase asociación**. A continuación se explica cada una de ellas:

- **Herencia**: son jerarquías de generalización/especialización. Como ya hemos visto en otro capítulo, la herencia es el mecanismo que permite a una clase de objetos (**subclase**) compartir la estructura y el comportamiento de la clase de objetos (**superclase**) de la que está heredando y tener nuevas estructuras y comportamiento independientes de la superclase.

  La herencia se puede obtener generalizando, es decir, extrayendo la superclase a través de las subclases. Por ejemplo, si tenemos las clases *libro* y *revista*, podemos extraer los elementos comunes y obtenemos la superclase *documento*. También puede ser a la inversa, es decir, a partir de la superclase, descomponer ésta en subclases con atributos y operaciones disjuntas. Siguiendo con el ejemplo de *documento*, esto ocurre si primero hubiéramos modelado la clase documento y luego la hubiéramos descompuesto en las subclases *libro* y *revista*.

  La Herencia se representa como una línea continua acabada en una punta de flecha hueca en el extremo de la superclase y orientada hacia ella. La figura 7.7 muestra un ejemplo de una relación de herencia.

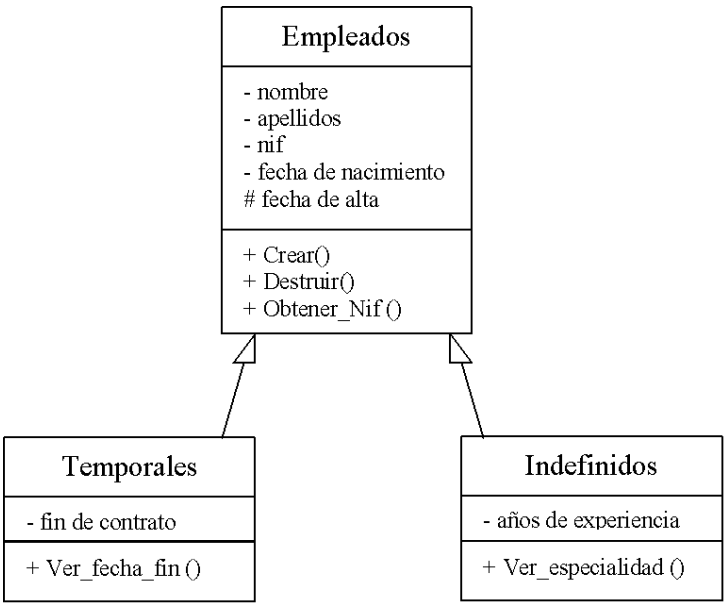


Figura 7.7: Ejemplo de relación de herencia

- **Asociación**: es el tipo de relación estructural más general y representa una dependencia semántica, es decir, las instancias de una clase están enlazadas con las instancias de otra clase. Por ejemplo, la clase *empleados*

puede tener una asociación con una clase *empresa* llamada *trabaja en*. Se representa como una línea continua, entre las clases asociadas, como se ve en la figura 7.8.

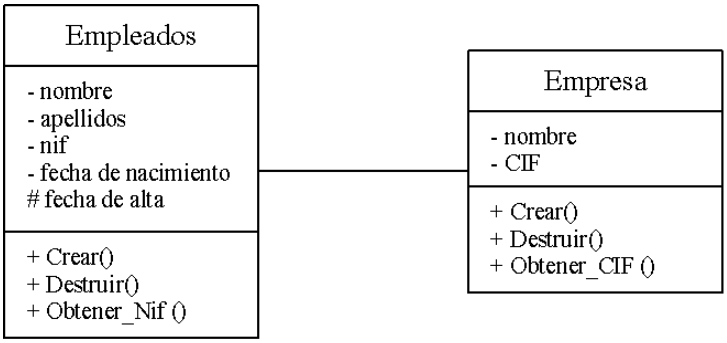


Figura 7.8: Asociación entre Empleados y Empresa

En una asociación se pueden representar elementos adicionales para aumentar el detalle del tipo de relación. De entre estos elementos vamos a destacar el **nombre** y la **multiplicidad**. El primero es el nombre de la asociación y debe ser lo suficientemente descriptivo como para distinguir esa asociación de cualquier otra. Se representa escribiendo el nombre junto a la línea de la asociación. La multiplicidad describe la cardinalidad de una relación, es decir, especifica cuántas instancias de una clase están asociadas con instancias de la otra y puede ser un número concreto, un rango o un conjunto de números, donde n o * representan cualquier valor. Se representa al lado de la clase. En la figura 7.9 se observa el nombre *trabaja en* y la cardinalidad, * para *empleados*, 1 para *empresa*.

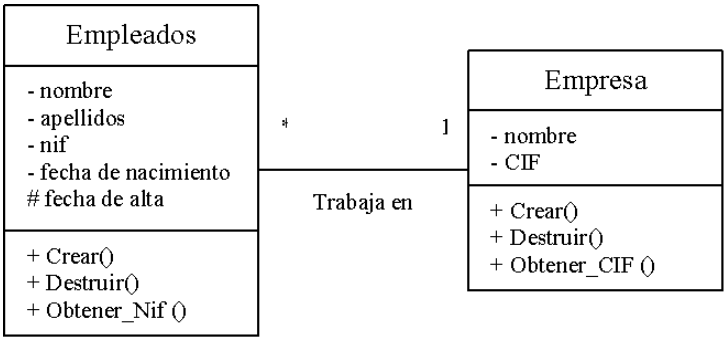


Figura 7.9: Asociación detallada entre Empleados y Empresa

Una asociación puede enlazar a una misma clase, lo que representa que

una instancia de esa clase está asociada a otras instancias de la misma clase. Por ejemplo, es fácil intuir que en *empresa* van a existir *empleados* que son *jefes* de otros *empleados*. La representación de este tipo de relación se muestra en la figura 7.10.

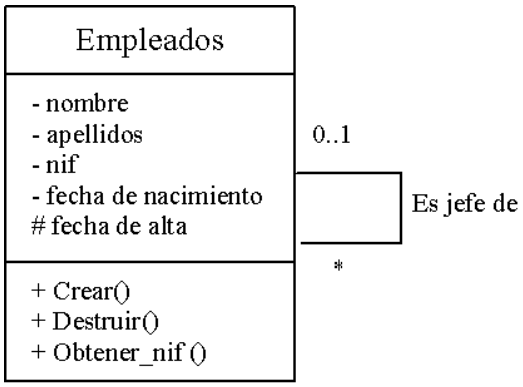


Figura 7.10: Ejemplo de asociación de una clase consigo misma

Dentro de las asociaciones, también existen propiedades avanzadas de ellas, de entre las que vamos a destacar dos: la **composición** y las **clases asociación**.

- **Composición**: es un tipo especial de asociación entre una clase, que representa la totalidad de esa clase, y las partes que la componen, con una fuerte relación de pertenencia, y, donde las instancias de las clases *parte* viven y mueren con la instancia del todo. En este tipo de relación, la clase que representa el todo, es la encargada de crear y destruir las instancias de las clases parte de ella. Además, las clases parte con una multiplicidad variable se pueden crear después de haberse creado la clase compuesta, pero a partir de ese momento, su ciclo de vida será el mismo que el de la clase compuesta. Por ejemplo, si pensamos en la clase *coche*, ésta puede estar compuesta de las clases *motor*, *ruedas*, *chasis* y *carrocería*, y no tiene sentido instanciar independientemente ninguna de las clases parte, si no dentro de la clase *coche*.

  La Composición se representa como una línea continua acabada en rombo sombreado en el extremo del todo. La figura 7.11 muestra un ejemplo de una relación de composición, en este ejemplo se trata de crear un jugador para un juego de baloncesto, esta clase *jugador baloncesto* realiza acciones complejas combinando las operaciones simples ofrecidas por los dos clases que lo componen, *piernas* y *brazos*.

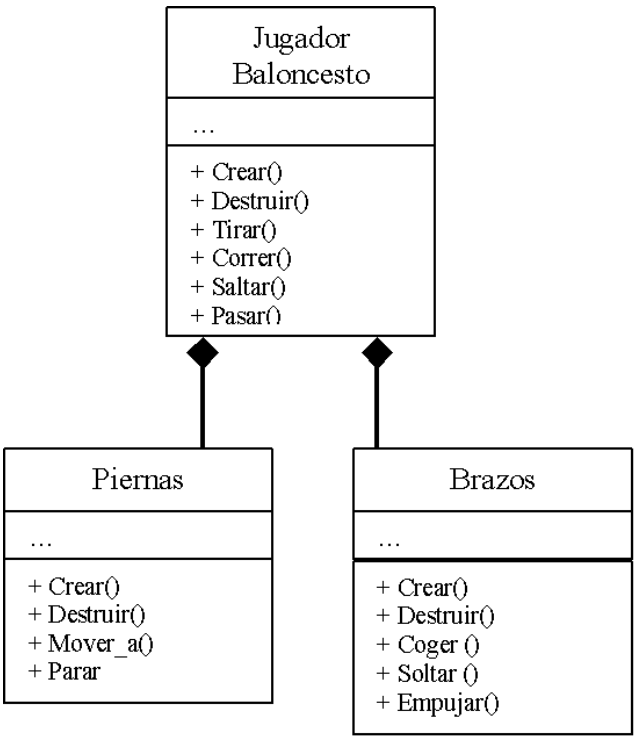


Figura 7.11: Ejemplo de relación de composición

- **Clases asociación**: se utilizan cuando necesitamos mantener propiedades y operaciones de una asociación. El concepto que representa se puede definir de dos maneras: como una asociación con propiedades de clase o como una clase que tiene propiedades de asociación. Por ejemplo pensemos en las clases *alumnos* y *asignaturas*, entre ellas existe una asociación *notas* que representa propiedades como *curso*, *puntuación* y *número de convocatoria*. Si intentáramos modelar *notas* como una clase normal, asociada a *asignaturas* por un lado, y a *alumnos* por otro, no estaríamos expresando lo mismo, ya que una instancia específica de *notas* no estaría ligada a un par específico de instancias de *alumnos* y *asignaturas*. Con este tipo de clases estamos representando que no puede existir esa asociación sin las clases que relaciona.

  Se representa con una línea discontinua desde la asociación hasta el símbolo de clase, como se representa en la figura 7.12.

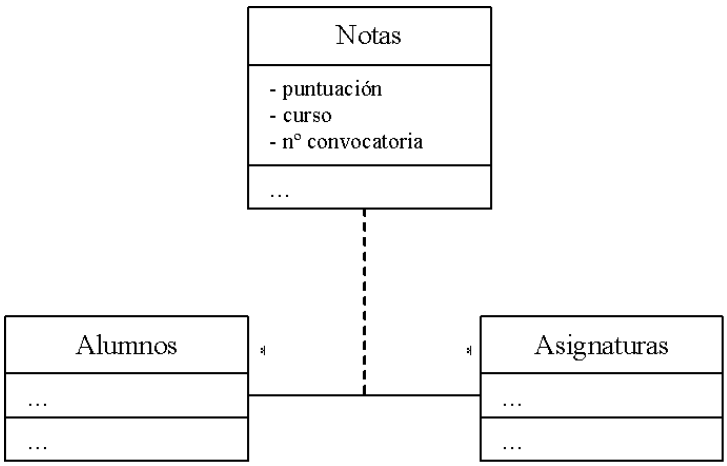


Figura 7.12: Clase Asociación Notas entre Alumnos y Asignaturas

7.4.3. Diagrama de Clases del Problema Propuesto

Ahora que ya sabemos cómo realizar un diagrama de clases vamos a aplicarlo al problema de la empresa de alquileres. Primero vamos a modelar las clases obtenidas en el apartado 1 del capítulo, junto con los atributos del apartado 2 y las operaciones del apartado 3. El resultado se observa en la figura 7.13.

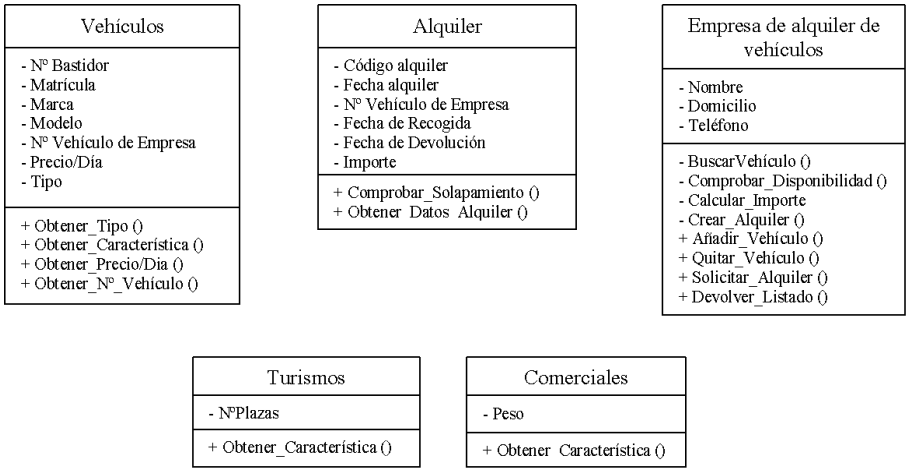


Figura 7.13: Clases del problema de empresa de alquiler de coches

Una vez obtenidas las clases, el siguiente paso debe ser estudiar las posibles relaciones que existen entre ellas. Una forma bastante sencilla de empezar

es buscando relaciones de herencia, que además, se deben haber descubierto normalmente en la fase de identificación de clases. Así, en nuestro ejemplo ya decidimos que había una clase, *Vehículos*, que tenía dos subclases que heredaban de ella, *Turismos* y *Comerciales*. La representación de esta relación de herencia se presenta en la figura 7.14.

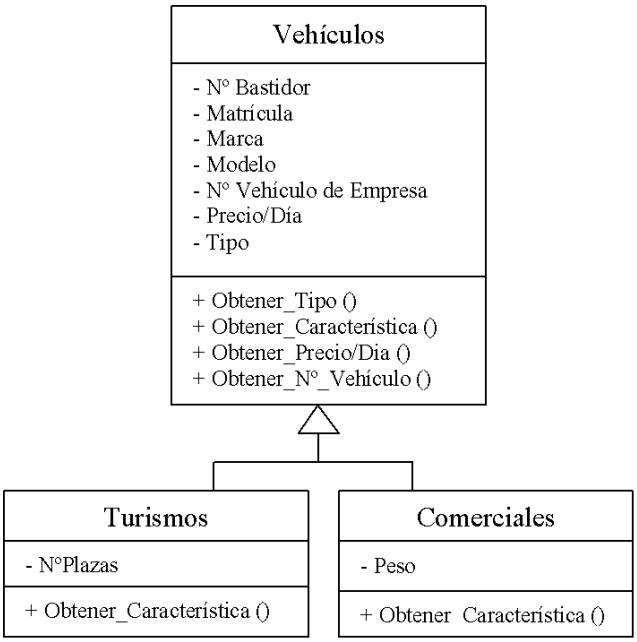


Figura 7.14: Relaciones de herencia de nuestro diagrama

El siguiente paso es estudiar si existe alguna relación de composición. Como ya hemos explicado anteriormente, una relación de composición es una relación con una fuerte relación de pertenencia donde la vida de las partes, una vez creadas por la clase compositora, coincide con la del todo. En nuestro ejemplo puede observarse que existe una relación de composición entre *Vehículos* y *Empresa de alquiler de vehículos*, ya que sólo tiene sentido que los *Vehículos* existan como parte de *Empresa de alquiler de vehículos*. La cardinalidad de la composición es de *n* en la parte de los *Vehículos* y de *1* en la parte de *Empresa de alquiler de vehículos*. En la figura 7.15 se ilustra esta relación.

En esta composición además sucede algo especial, ya que, los vehículos que tiene la *Empresa*, son ofrecidos para alquilar. Cada alquiler, como ya vimos en el apartado 1, será una instancia de la clase *Alquiler*, pero esta clase no tiene ningún sentido en el sistema si no existieran las clases *Vehículos* y *Empresa de alquiler de vehículos*. Esto es debido a que *Alquiler* es una clase asociación. Por tanto, una vez que ya sabemos como situar la clase *Alquiler* en el diagrama,

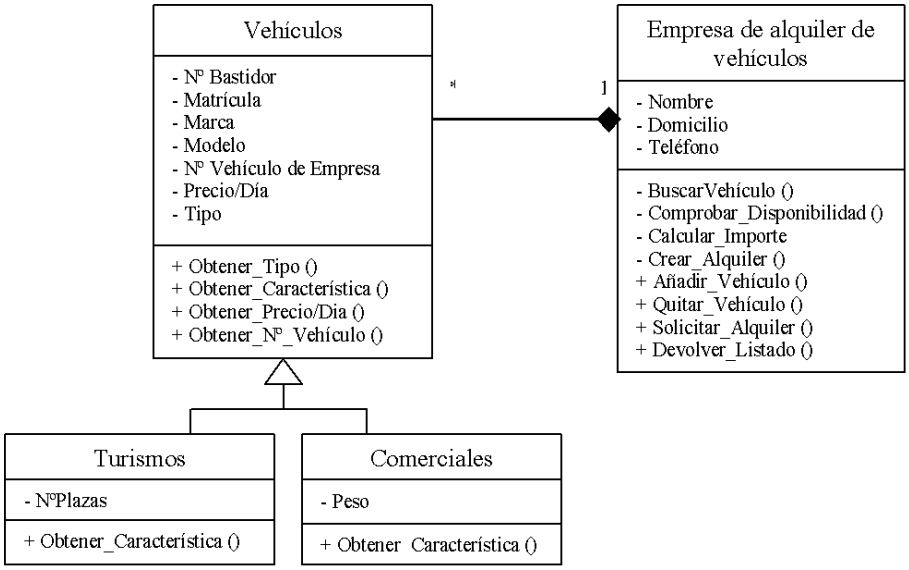


Figura 7.15: Relación de composición de nuestro Diagrama

éste quedaría definitivamente como se muestra en la figura 7.16.

7.5. Modelado de aspectos dinámicos

El **diagrama de secuencias** representa, para cada caso de uso del sistema, las diferentes interacciones que se producen entre los objetos de éste, destacando la ordenación temporal de los mensajes que se envían. Es decir, sirve para ayudar a comprender mejor los casos de uso y observar la comunicación entre objetos. Además, ayudan a encontrar lagunas en las especificaciones realizadas. Se compone de:

- Los **objetos** y **actores participantes**, que se colocan horizontalmente en la parte superior. Se suele colocar el actor que inicia el caso de uso en el extremo de la izquierda. Para representar una instancia de objeto, se escribe el nombre de la instancia seguido de dos puntos y el nombre de la clase, todo subrayado. Podemos indicar instancias anónimas dejando en blanco el nombre de la instancia. Puede interesarnos usar una instancia anónima para representar distintas instancias de una misma clase a la que están enviando el mismo mensaje desde un bucle.
- El **tiempo de vida** de los participantes, representado con una línea discontinua vertical, que tiene su origen en los objetos y actores, y que representa el avance del tiempo de arriba hacia abajo. La mayoría de

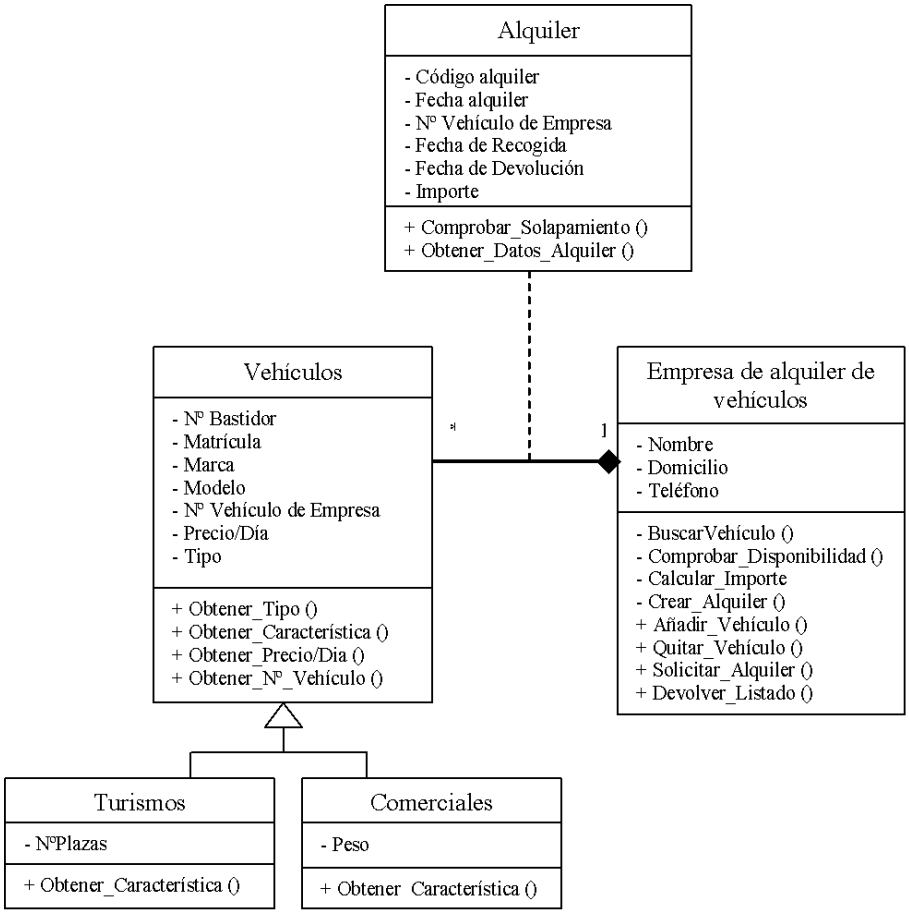


Figura 7.16: Diagrama de clases del problema propuesto

los objetos participantes normalmente tendrán una línea de arriba hacia abajo, pero puede ocurrir que una instancia se cree o se destruya durante las interacciones. En el primer caso, el tiempo de una instancia comenzaría al recibir el mensaje **create**, y en el segundo, la destrucción de una instancia, el tiempo terminaría al recibir un mensaje **destroy** y, además, se representa con una gran X donde concluya su línea de tiempo.

- Las **interacciones**, dibujadas como flechas horizontales etiquetadas entre las líneas de tiempo de cada objeto y orientadas del emisor al receptor. Nunca representan flujos de datos, sino mensajes o eventos fundamentales para la interacción. A partir de la primera interacción, que se coloca en la parte superior, deben situarse el resto de interacciones de arriba a abajo para indicar el orden temporal de cada una. Cuando queremos representar la respuesta de una interacción, podemos dibujarla como una flecha discontinua etiquetada orientada, en este caso, del receptor hacia el emisor que generó la interacción.
- Los **focos de control**, son rectángulos delgados sobre la línea de tiempo, y representan el periodo durante el cual ese objeto tiene el control del sistema, bien porque está ejecutando él mismo una acción, o bien porque lo está ejecutando algún procedimiento subordinado. Así, si existe una jerarquía de llamadas, el control no se libera hasta que se haya completado el último retorno. La parte superior e inferior del rectángulo deben estar alineadas con el comienzo y fin de una acción.

En la figura 7.17. se presenta el esquema de un diagrama de secuencias con los componentes mencionados.

7.5.1. Bifurcaciones e iteraciones

Un único diagrama normalmente sólo debería mostrar un flujo concreto de control. De hecho, existen otros diagramas dentro de UML que permiten especificar mejor este tipo de estructuras de control de flujo. No obstante, en el caso de usar diagramas de secuencia, es recomendable representar las bifurcaciones e iteraciones utilizando un diagrama de secuencias por cada posibilidad a expresar.

De todos modos, si decidimos representar los distintos flujos de control dentro de un solo diagrama de secuencia, se pueden representar tanto la bifurcación como la iteración, de la siguiente forma:

- La **bifurcación** como dos flechas de interacción saliendo del mismo origen, y, cada una, con la condición para que se ejecuten al principio de la flecha entre corchetes. También es posible representar de la misma manera, la posibilidad de que una única interacción, se ejecute o no, simplemente poniendo la condición de ejecución en el origen de la flecha entre corchetes.

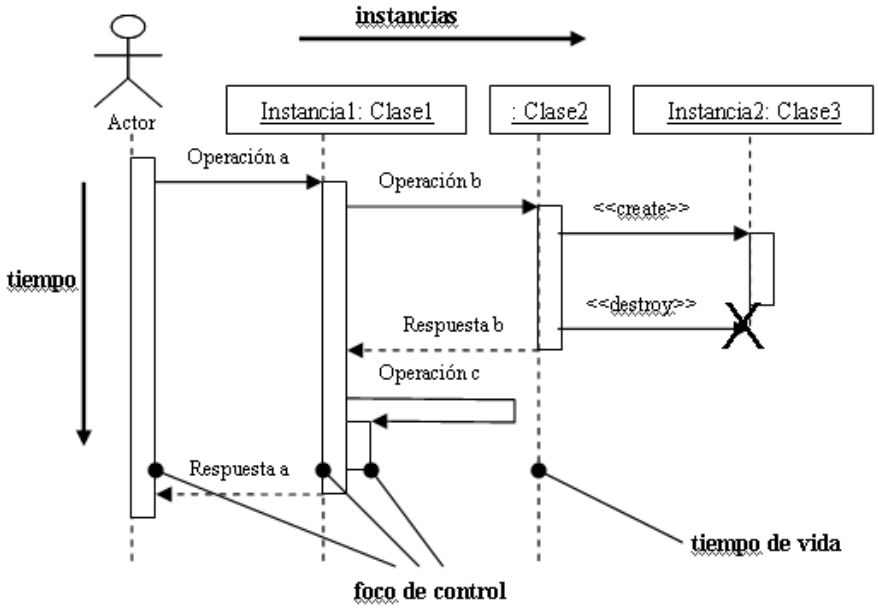


Figura 7.17: Esquema de un Diagrama de Secuencias

- Para representar una **iteración** se coloca un * delante de la interacción junto con la condición entre corchetes para seguir iterando. Cuando es más de una interacción consecutiva la que se va a iterar, es decir, tenemos un bucle con distintas interacciones, es recomendable representar todas las operaciones del bucle dentro de un rectángulo, junto con la condición de iteración en la parte superior izquierda

En la figura 7.18 se muestra el esquema de un diagrama de secuencias con las nuevas opciones mencionadas.

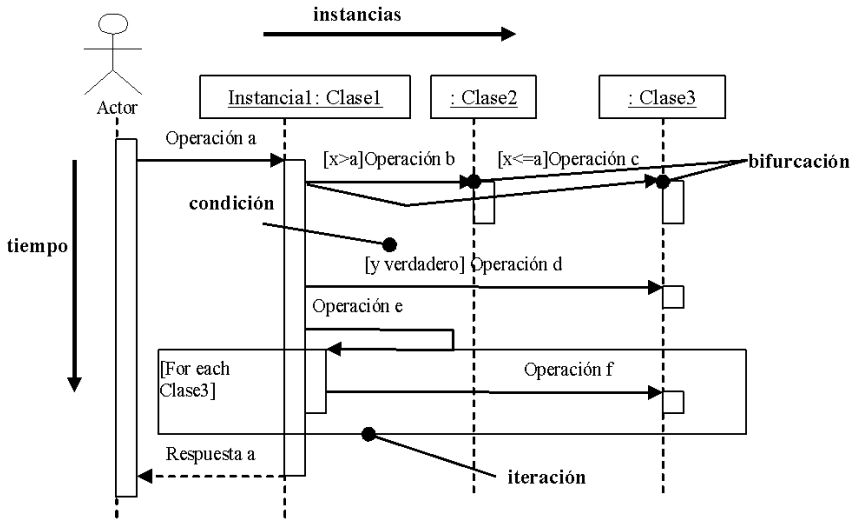


Figura 7.18: Esquema de Diagrama de Secuencias con condiciones, bifurcaciones e iteraciones.

7.5.2. Diagrama de Secuencias de una operación del Problema Propuesto

En este apartado sólo vamos a representar el diagrama de secuencias del caso de uso *Solicitar Alquiler*, que estará disponible para los clientes del sistema. Los otros tres casos de uso, *Añadir Vehículos*, *Quitar Vehículo* y *Devolver Listado*, son bastante más sencillos que el que hemos elegido, y por eso, animamos al lector a que lleve a cabo su realización tras finalizar la lectura de este apartado.

Como ya comentamos al identificar las operaciones de nuestras clases, *Solicitar Alquiler* es un método de la clase *Empresa de Alquiler de Vehículos* que ejecutará varias operaciones dentro del sistema y que se explica detalladamente a continuación.

En primer lugar, debe recorrer las instancias de *Vehículos*, buscando uno que sea del tipo y característica solicitado, y que esté disponible en las fechas pedidas. Para realizar este primer cometido, llama a la operación privada *Buscar Alquiler*. Esta operación se compone de una serie de llamadas iterativas que, en primer lugar, se encaminan a encontrar una instancia de *Vehículos* con las propiedades requeridas (operaciones públicas de la clase *Vehículos*, *Obtener Tipo* y *Obtener Característica*), y, en segundo, a comprobar si la instancia de *Vehículo*, en caso de encontrar una, está disponible en las fechas pedidas. Para comprobar que un vehículo está libre en las fechas solicitadas, *Buscar Vehículo* llama a la operación privada *Comprobar Disponibilidad*, que deberá recorrer todas las instancias de la clase *Alquiler*, chequeando que el vehículo encontrado

está libre en las fechas requeridas (operación *Comprobar Solapamiento* de la clase *Alquiler*).

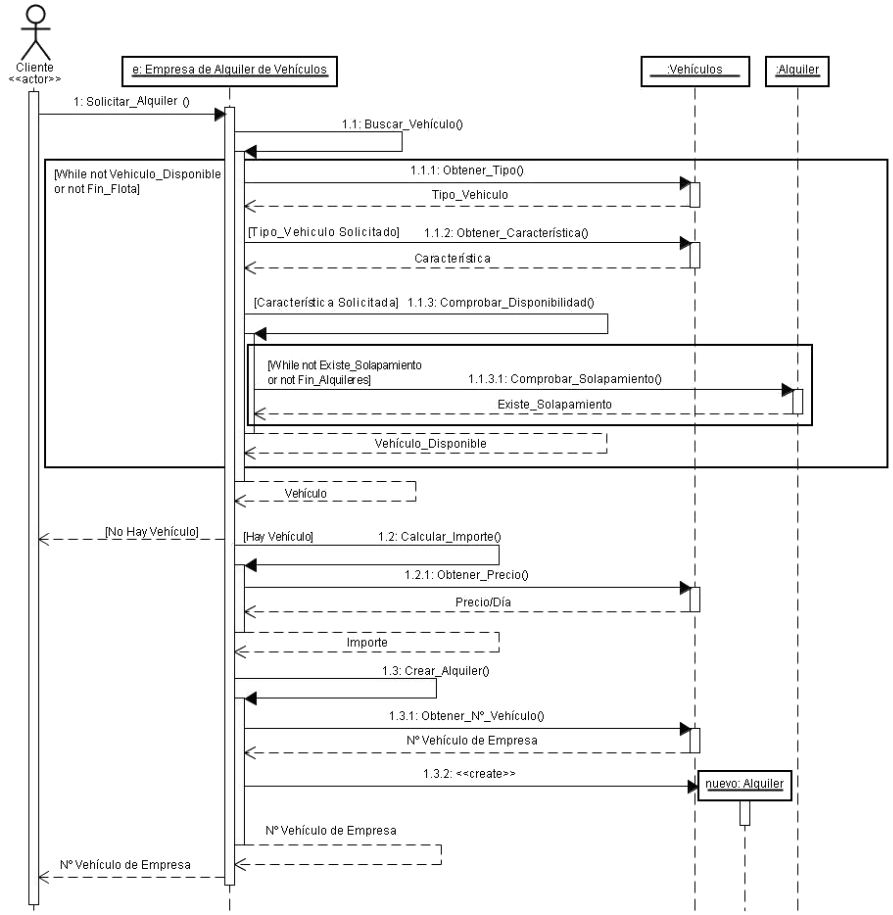


Figura 7.19: Diagrama de Secuencias de Solicitar Alquiler

En caso de no encontrarse una instancia de vehículo con las características solicitadas, o de estar las instancias encontradas ocupadas, *Buscar Alquiler* debe comunicar la situación a *Solicitar Alquiler* para que le indique al cliente que no es posible realizar la operación.

Si *Buscar Alquiler* encuentra una instancia de *Vehículos* que cumpla todos los requisitos, *Solicitar Alquiler* debe continuar para formalizar el alquiler. Para este segundo cometido, llama a la operación privada *Calcular Importe*. Ésta, obtendrá el coste del alquiler a partir de la duración del mismo, y del precio del vehículo encontrado (operación privada de *Vehículos, Obtener Pre-*

cio Día). Una vez que *Solicitar Alquiler* tiene el importe del alquiler, llama a la operación *Crear Alquiler* para concretar la operación. *Crear Alquiler* llama en primer lugar a la operación privada de *Vehículos*, *Obtener Nº Vehículo*, y a continuación crea una nueva instancia de *Alquiler* con todos los datos necesarios. Para terminar, *Solicitar Alquiler* devuelve al cliente el *Nº de Vehículo* que ha alquilado.

El diagrama de secuencias de toda la operación *Solicitar Alquiler* se muestra en la figura 7.19

7.6. Resumen

En este capítulo se ha presentado una propuesta de identificación de clases y se ha introducido UML como notación de modelado. La identificación de clases no es algo trivial, al contrario, es una tarea crítica en el diseño de un sistema, ya que si partimos de una mala identificación, este defecto se va a ir arrastrando hasta el sistema final implementado. Como hemos visto, el proceso de identificación parte de un análisis gramatical del problema para la obtención de una serie de candidatos a ser clases o atributos (sustantivos), y a ser operaciones o relaciones (verbos). Es conveniente no quedarse sólo en el análisis gramatical, ya que diferentes estilos de escritura pueden llevar a diferentes modelados. Por eso, el siguiente paso debe ser evaluar los candidatos obtenidos, aceptándolos o rechazándolos según una serie de criterios. En el caso de la identificación de operaciones, hemos visto que existe una herramienta denominada Diagrama de Casos de Uso que puede sernos muy útil. Hay que señalar que puede ser igual de nefasto para el sistema olvidarse de alguna clase o una operación como tener clases u operaciones inadecuadas.

A lo largo de todo el capítulo, se ha utilizado un problema de un posible sistema real, que se ha modelado para ilustrar todos los métodos presentados. Así, una vez obtenida la identificación de clases, atributos y operaciones, hemos visto como se puede representar gráficamente el modelo obtenido a nivel estático y dinámico, con la ayuda del diagrama de clases, en el primer caso, y, del diagrama de secuencias y de casos de uso en el segundo.

cc BY NC SA

Capítulo 8

Otras características de C++

En este capítulo se realiza una breve exposición de características del lenguaje C++ que no están directamente relacionadas con el paradigma de programación orientada a objetos, pero que tienen gran relevancia dentro del lenguaje. La finalidad de este apartado es servir de punto de partida hacia una mayor profundización en el lenguaje C++.

8.1. Programación genérica

La **genericidad** es una construcción interesante en un lenguaje orientado a objetos. No es de uso exclusivo en este tipo de lenguajes, pero en ellos adquiere un significado más completo. La genericidad es uno de los conceptos más importantes utilizados en la reutilización; permite la utilización de tipos como parámetros en la definición de una clase o función, de forma que no requiere la declaración explícita del tipo de algún parámetro o parámetros; y así, permite que esta clase o función pueda adaptarse a distintos tipos de datos en su utilización.

La principal ventaja de la utilización de **plantillas** en funciones y clases es poder utilizarlas para distintos tipos de datos sin modificar la definición de dicha función o clase. Normalmente las plantillas se declaran en un fichero cabecera.

Veremos a continuación el formato de definición de plantillas de funciones y de clases así como ejemplos de usos de ambos tipos.

8.1.1. Formato de plantilla de función

Una plantilla de función se define escribiendo la palabra reservada **template** seguida del nombre de la función. Si la función puede recibir como parámetros distintos tipos de datos, o si el tipo de la función es un tipo no conocido previa-

mente; la definición de la función se realizará escribiendo la palabra reservada **template** seguida de los tipos de parámetros genéricos de que consta la función separados por coma y entre $< >$. Cada parámetro se indica con la palabra clave **class** y un identificador, como se muestra a continuación:

```
template <parámetros plantilla>
declaración de función
```

Veamos algunos ejemplos de plantillas de función:

```
template <class T>
T func1 (T a, int b) {
   // cuerpo de la función }
template <class T>
T func2 (T a, T b) {
   // cuerpo de la función }
template <class T1, class T2>
T2 func3 (T1 a, T2 b) {
   // cuerpo de la función }
```

La función *func1* recibe como parámetros *a* (de un tipo no definido previamente) y *b* de tipo *int*. A su vez, esta función es del mismo tipo que el tipo del parámetro *a* (definido como *T*). Este tipo se conocerá cuando se realice la llamada a la función desde el programa principal.

La función *func2*, por su parte recibe dos parámetros del mismo tipo (aún no definido) y a su vez esta función es del mismo tipo que estos dos parámetros.

Por último, la función *func3* recibe como parámetros *a* y *b*, siendo ambos de distinto tipo. Esta función devolverá un valor de un tipo igual al del parámetro *b* (en el ejemplo identificado como *T2*).

Como puede observarse en estos tres ejemplos, los parámetros de la función de un tipo no definido pueden utilizarse con otros parámetros de tipos definidos (int, char, etc.) o con parámetros de otro tipo no definido distinto (*T1* y *T2* por ejemplo). La función por su parte puede devolver cualquier tipo de datos, incluyendo los tipos utilizados en la lista de parámetros.

Podríamos definir como ejemplo sencillo una función llamada *mayor* que devolviera el mayor de los dos parámetros que recibe la función. La función devolvería un valor de un tipo igual al tipo de los dos parámetros que recibe como entrada:

```
template <class Tipo>
Tipo mayor (Tipo a, Tipo b) {
  if (a > b) return a;
  return b;
}
```

Veamos un ejemplo a continuación de la llamada a la función *mayor* utilizando distintos tipos de parámetros de entrada. En primer lugar se llamará a la función con dos parámetros de tipo *int* (x e y) y a continuación, se llamará a la misma función utilizando dos parámetros de tipo carácter (a y b). Como puede observarse la forma de llamar a la función es igual a la forma en que se realizaría la llamada a una función definida normalmente.

```
void main() {
   int x=4, y=6;
   char a='g', b='e';
   //llamada a la función con parámetros de tipo entero
   cout << "El mayor es: " << mayor(x,y);
   //llamada a la función con parámetros de tipo carácter
   cout << "El mayor es: " <<  mayor(a,b);
  }
```

8.1.2. Plantillas de clase

Cuando una clase se define como plantilla, esta clase podrá trabajar con distintos tipos de datos. Una de las aplicaciones más importantes del uso de clases como plantillas es la implementación de contenedores, es decir, clases que contienen objetos de un mismo tipo de datos, como pilas, colas, listas, árboles, etc. Podremos, por ejemplo, implementar una clase *Cola* para crear una estructura de datos cola en la que se puedan almacenar enteros, cadenas de caracteres, etc. sin tener que implementar una clase *Cola* para cada uno de estos tipos distintos.

El formato de una plantilla de clase es muy sencillo, se utiliza la palabra clave **template** seguida de la lista de los tipos de datos genéricos separados por coma y precedidos por la palabra clave **class** y encerrados entre '$<$ $>$'. A continuación se especifica el nombre de la clase precedido, como se hacía normalmente, por la palabra clave **class**.

```
template <class nombre_tipo>
class nombre_clase {
   // definición de la clase
}
La definición de una pila genérica podría ser:
template <class Tipo>
class Pila {
   // definición de la clase
}
```

Como puede observarse, los elementos que se van a almacenar en la pila pueden ser de distinto tipo. La definición de un objeto de tipo pila que almacene enteros se realizaría de la siguiente forma:

```
Pila <int> pilaenteros; //pila de números enteros
Pila <float> pilareales; //pila de números reales
```

El formato de definición del constructor de una clase genérica sería de la siguiente forma:

```
template <parámetros plantilla >
nombre_clase <parámetros plantilla> :: nombre_clase
```

El formato de definición de una función miembro de una clase genérica sería por su parte de la siguiente forma:

```
template <parámetros plantilla >
tipo_resultado nombre_clase <parámetros plantilla> ::
         nombre_de_función_miembro (declaraciones parámetros)
```

Para finalizar este apartado, se muestra al lector un ejemplo completo de implementación de una clase genérica que define el comportamiento de una estructura de datos cola.

```
template <class Tipo>
class Cola {
   typedef struct Nodo
   {
      Tipo dato;
      Nodo *sig;
   }
   tiponodo *frente, *final;
   int numElementos;
public:
   Cola ();
   bool vacia(); typedef struct
   void insertar(Tipo valor);
   Tipo frente();
   void borrarFrente();
   ~Cola();
};
template <class Tipo>
Cola<Tipo>::Cola() {
   frente=final=NULL;
   numElementos=0;
}
template <class Tipo>
bool Cola<Tipo>::vacia() {
   return (frente == NULL);
```

```
}
template <class Tipo>
void Cola<Tipo>::insertar(Tipo valor) {
   tiponodo *nuevonodo = new tiponodo;
   nuevonodo->dato=valor;
   nuevonodo->sig = NULL;
   if (vacia())
      frente=nuevonodo;
   else
      final->sig=nuevonodo;

   final=nuevonodo;
   numElementos++;
}
template <class Tipo>
void Cola<Tipo>::borrarFrente(){
  if (!vacia())
  {
    tiponodo* aux;
    aux=frente;
    frente=frente->sig;
    delete frente;
    numElementos--;
  }
}
template <class Tipo>
Cola<Tipo>::~Cola() {
  Cola *aux;
  while(!vacia())
    borrarFrente();
  fin=NULL;
}
int main() {
  int i;
  Cola <int> colaenteros; //cola de enteros
  for(i=0;i<10;i++)
    colaenteros.insertar(i);

  cout<<colaenteros.frente();
  colaenteros.borrarFrente();
  cout<<colaenteros.frente();
  Cola <char> colacaracter; //cola de caracteres
  colacaracter.insertar('h');
  colacaracter.insertar('o');
```

```
    colacaracter.insertar('l');
    colacaracter.insertar('a');
    cout<<colaenteros.frente();
    colaenteros.borrarFrente();
    cout<<colaenteros.frente();
    return 0;
}
```

8.2. Biblioteca estándar

El lenguaje C++ incluye una biblioteca básica que da apoyo a los entornos de desarrollo estándar: la **biblioteca estándar de C++**. Esta biblioteca constituye la base de desarrollo para los sistemas que se crean mediante este lenguaje. Entre otras aportaciones, esta biblioteca ofrece:

- Internacionalización.
- Soporte para gestión de memoria y para la información de tipo en tiempo de ejecución.
- Funciones que no pueden implementarse optimamente mediante el lenguaje.
- La biblioteca estándar de C.
- Cadenas y flujos de E/S.
- Diagnóstico: excepciones.
- Estructuras de datos y algoritmos genéricos.
- Soporte para el cálculo numérico.
- Proporciona una base común para otras bibliotecas.

A lo largo de este apartado se introduce someramente las principales utilidades de la **Biblioteca Estándar de C++**. En primer lugar, se aclara el concepto de espacio de nombres. Posteriormente, se presentan las cadenas estándar: **string**. Y, finalmente, se explica brevemente la **Biblioteca de Plantillas Estándar** (Standard Template Library, STL). También, es conveniente apuntar que los flujos de entrada y salida (**cout**, **cin**, **cerr**) usados durante todo el libro forman parte de la librería estándar.

8.2.1. Espacios de nombres

Los **espacios de nombres** pueden verse como *agrupaciones lógicas* de declaraciones, es decir, si algunas declaraciones pertenecen a un mismo grupo pueden incluirse en el mismo espacio de nombres. Los espacios de nombres son un recurso introducido por C++ para evitar el problema que ocurría en C cuando los programas eran de gran tamaño. En estos casos, se presentaban problemas de colisiones entre identificadores de objetos o subprogramas que utilizaban nombres iguales. En C++, con la utilización de espacios de nombres se puede evitar este problema divididiendo el espacio total de los identificadores de objetos del programa en subespacios distintos e independientes. La palabra clave utilizada para definir un espacio de nombres es **namespace**.

Un espacio de nombres se define escribiendo **namespace** seguido del nombre y delimitando el espacio de nombres entre llaves. Veamos un ejemplo de definición de espacios de nombres:

```
namespace empresa1 {  //definición de espacio de nombres
   int codigo;
   float obtener_importe( ) { return valor };
}
namespace empresa2 {  //definición de espacio de nombres
   int codigo;
   float obtener_importe( ) { return valor };
}
empresa1::codigo=100;  //acceso al codigo de empresa1
cout<<empresa1::obtener_importe();
empresa2::codigo=200;  //acceso al codigo de empresa2
cout<<empresa2::obtener_importe();
```

En este ejemplo se han definido dos espacios de nombres: *empresa1* y *empresa2*. Como podemos observar, los dos espacios utilizan variables y funciones con el mismo nombre y no existe ningún problema, ya que ambos espacios de nombres trabajan de una forma totalmente independiente. La forma de acceder a los objetos declarados dentro de cada espacio de nombres es anteponiendo al nombre del objeto al que se desea acceder el nombre del espacio de nombres seguido de **::** (en el ejemplo: *empresa1::codigo=100*).

El espacio de nombres en el que están definidas las utilidades de la librería estándar se denomina **std**. A la hora de utilizar la biblioteca estándar, normalmente, el espacio de nombres, seguido del operador de alcance, debe anteponerse a cualquier definición de variable de un tipo que esté incluido en él. Por ejemplo, a continuación se muestra un ejemplo de utilización muy simple del tipo **string**, que, como se verá, está incluido en la librería estándar de C++ y se utiliza para trabajar con cadenas de caracteres y realizar operaciones entre ellas, un ejemplo de su uso podría ser el siguiente:

```
#include <string>
#include <iostream>
void main( ) {
  std::string cade="Prueba de cadena de caracteres";
  std::cout << cade;
}
```

Como puede verse en el ejemplo, se define una variable *cade* de tipo **string** y se muestra por pantalla dicha variable. Se antepone el espacio de nombres a la definición del tipo **string** y a la instrucción **cout**, utilizando el prefijo **std::**. El identificador **std** hace referencia al espacio de nombres estándar. Existe otra forma de utilizar un espacio de nombres que en este ejemplo sería útil. Esta forma consiste en declarar el espacio de nombres que se va a utilizar de una forma global, y así, no sería necesario anteponerlo en cada acceso o definición de variable; esta es la forma más normal de su utilización. Puede modificarse el ejemplo anterior declarando el espacio de nombres globalmente, el resultado es el siguiente:

```
#include <string>
#include <iostream>
using namespace std;
void main( ) {
  string cade="Prueba de cadena de caracteres";
  cout << cade;
}
```

En este ejemplo, puede observarse que de esta forma se complica menos el código.

En los ejemplos mostrados en los capítulos anteriores de este libro ha sido eliminada la utilización explícita de **include** y **namespace** con la intención de que el lector pudiera leer y comprender el código de una forma más clara.

8.2.2. El tipo string

En este apartado nos detendremos en el tipo **string**. Este es uno de los tipos de datos más importantes que son definidos en la librería estándar de C++. La ventaja principal que ofrece el uso del tipo **string** es evitar los problemas de manejo de cadenas de caracteres que existían en C, como declarar las cadenas de tipo puntero a carácter, reservar memoria para variables de este tipo, controlar el fin de cadena, etc. El tipo **string** ofrece las siguientes operaciones de manipulación de cadenas:

- asignación
- comparación

- concatenación

Veremos en este apartado algunos ejemplos sencillos de realización de estas operaciones para mostrar su utilización al lector.

La **asignación** de cadenas se realiza de una forma muy sencilla, simplemente se iguala una cadena a la otra utilizando el operador =. De esta forma, la cadena asignada se copia y después de la asignación existen dos cadenas separadas con el mismo valor. Esta operación se realiza utilizando la sobrecarga de operadores, en este caso, el operador = ha sido sobrecargado para poder ser utilizado en asignación de cadenas. Un ejemplo de asignación de cadenas sería el siguiente:

```
#include <string>
using namespace std;
void asigna( ) {
  string cade1="Valor1";
  string cade2="Valor2";
  cade1 = cade2;
  cade2[5] = 'x';
  cout << cade1;
  cout << cade2;
  string cade3='x'; //Error, no permitido
}
```

Cuando se realiza la asignación, *cade1* y *cade2* son dos cadenas distintas con el mismo valor (*Valor2* en nuestro ejemplo). Cuando se modifica la posición 5 de la segunda variable, *cade1* seguirá conteniendo la cadena *Valor2* y *cade2* contendrá la cadena *Valorx*. Como puede verse en este ejemplo, una cadena no podrá inicializarse mediante un solo carácter, pero la asignación de un solo carácter a una cadena sí está permitida.

La **comparación** entre cadenas puede realizarse entre cadenas de su propio tipo y con arrays de caracteres del mismo tipo carácter. La forma más sencilla de comparar dos cadenas es utilizando los operadores normales de comparación (==, !=, >, <, >= y <=) que han sido sobrecargados para poder utilizarlos en cadenas. Un ejemplo sencillo de comparación de cadenas es el siguiente:

```
#include <string>
using namespace std;
void compara(const string& cade1, const string& cade2) {
  if(cade1 == cade2)
   cout << "Cadenas iguales";
  if(cade1 == "Valor1" || cade1 == "Valor2")
    cout << "Comparando una cadena con un valor determinado";
}
```

La **concatenación** de cadenas se realiza utilizando el operador +. Este operador ha sido también sobrecargado para poder ser utilizado con cadenas de caracteres y el resultado de su aplicación es la construcción de una cadena a partir de dos cadenas colocando una detrás de la otra. Un ejemplo de concatenación de cadenas es el siguiente:

```
#include <string>
using namespace std;
string concatena(const string& cade1, const string& cade2) {
  cout << cade1 + '-' + cade2;
  return cade1 + '-' + cade2;
}
```

El resultado de esta función es mostrar por pantalla el contenido de la primera cadena y el de la segunda separados por un carácter '-' y devolver esto mismo como una única cadena de caracteres.

8.2.3. STL

Una de las principales razones del éxito de C++ es el hecho de que existe un gran número de bibliotecas disponible para facilitar el desarrollo de aplicaciones, ofreciendo componentes probados y de confianza. Una de las bibliotecas construidas con más cuidado es la Biblioteca de Plantillas Estándar (Standard Template Library, STL), desarrollada por Alexander Stepanov, Meng Lee y sus compañeros en Hewlett-Packard. Esta biblioteca fue aceptada como parte del estándar C++ ISO/IEC 14882.

La STL es una biblioteca multipropósito que se centra en las estructuras de datos y en los algoritmos. Utiliza profusamente el mecanismo de plantillas para crear componentes parametrizables. El diseño uniforme de los interfaces permite una cooperación flexible de los componentes y, además, la construcción de nuevos componentes al estilo STL. Por lo tanto, la STL es un marco de trabajo extendible y de uso universal, que ofrece muchas ventajas con respecto a la calidad, eficiencia y productividad. La STL está formada básicamente por una serie de contenedores, algoritmos e iteradores genéricos.

Un contenedor es un objeto que contiene otros elementos. Los contenedores en C++ se diseñaron para cumplir dos propósitos básicos: proporcionar la máxima libertad en el diseño de un contenedor individual y, a la vez, permitir que los contenedores presenten una interfaz común a los usuarios. Por su parte, los iteradores son clases que se pueden usar para navegar a través de los contenedores sin que el programador tenga que saber el tipo real usado para identificar los elementos. El cuadro 8.1 ofrece un resumen de los principales contenedores definidos en la STL.

La explicación del manejo de los contenedores, iteradores y algoritmos de la STL quedan fuera del alcance de este libro. Se recomienda al lector interesado en este tema que profundice en su estudio a través de la bibliografía aportada.

Contenedores STL
vector
deque
list
stack
queue
priority_queue

Cuadro 8.1: Contenedores

8.3. Funciones friend

Como se vió en el capítulo 3, las funciones que no eran miembro de una clase no podían acceder a los miembros privados de esa clase. En este apartado veremos que existe un caso en el que una función no miembro podrá acceder a los miembros privados de una clase. Este caso se produce cuando una función es definida como **friend** (afín). En algunas situaciones necesitaremos acceder a dos o más clases diferentes desde una función, en estos casos será útil la definición de funciones **friend**. Aunque este tipo de funciones puede ser útil en algunos casos, debe tenerse en cuenta que su uso está en contraposición con el concepto de encapsulación, por lo cual no es muy conveniente utilizarlas en casos en los que no sea totalmente necesario.

Veremos un ejemplo de su uso a continuación: supongamos que debemos comprobar si el nombre de un cliente es igual al usuario de un teléfono, debemos comparar entonces las dos variables *usuario* y *nombre* de las clases *Telefono* y *Cliente* (definida a continuación) respectivamente. Definiremos una función *comparaNombre* como **friend** que accederá a los miembros de las dos clases y devolverá un 1 si ambos nombres son iguales. La clase *Telefono* será igual a la utilizada anteriormente y la clase *Cliente* es una nueva clase que contiene cuatro variables miembro privadas a la clase y tres métodos públicos.

```
class Telefono;
class Cliente {
  string nombre;
  string dni;
  string direccion;
  string localidad;
public:
  //Función afín a la clase Cliente
  friend int comparaNombre (Telefono tel1, Cliente cli1);
  string obtenerDni();
};
class Telefono {
  string numero;
```


cc BY NC SA

```
  string tipo_tarifa;
  string usuario;
  float importe;
  void inicializarImporte( );
public:
  //Función afín a la clase Telefono
  friend int comparaNombre (Telefono tel1, Cliente cli1);
  void actualizarImporte(float _importe);
  float obtenerImporte( );
  void visualizarUsuario( );
  string &obtenerTipotel( );
};
int comparaNombre (Telefono tel1, Cliente cli1) {
  return (tel1.usuario == cli1.nombre);
}
```

Como puede observarse, no se utiliza ningún operador de ámbito ni de clase en la definición de la clase **friend**, ya que no es una función miembro. Es conveniente también destacar que aparece la definición de la clase *Telefono* justo antes de definir la clase *Cliente*, esto es debido a que, como en la función *friend* se utiliza como parámetro una variable *tel1* de la clase *Telefono*, es necesario hacer referencia a dicha clase que aún no ha sido definida para evitar errores. En este ejemplo concreto podría evitarse la utilización de la función *comparaNombre* definiendo dos funciones miembro en cada clase como públicas que devolvieran el usuario y nombre respectivamente y comparando ambos valores devueltos. Sin embargo, hay ocasiones en las que es necesario acceder a miembros privados de distintas clases, en estos casos, la opción de usar funciones afines es la más correcta.

8.4. Funciones en línea

Otro tipo de funciones utilizadas en la POO son las funciones **en línea**. Estas funciones se caracterizan porque están definidas totalmente en la declaración de la clase y porque el código de la función se expande en el punto donde se realiza la llamada a la misma.

En el ejemplo mostrado a continuación se ha definido la función *inicializarImporte* como una función en línea (**inline**) dentro de la clase *Telefono*. Como puede verse, el contenido de la función está definido dentro de la declaración de la clase *Telefono*; esa es la principal característica que nos hace diferenciar la función *inicializarImporte* como una función en línea.

Estas funciones son bastante eficientes, ya que la velocidad del programa aumenta, pero es aconsejable no utilizar demasiado funciones en línea o bien utilizar este tipo de funciones de pequeño tamaño, ya que consumen mucha

más memoria. De todas formas, el compilador se reserva la decisión final de que una función sea **inline** o no para evitar problemas de este tipo.

Con una función en línea, cada llamada de función en el programa se sustituye por una versión compilada de la definición de la función, por esto las llamadas a este tipo de funciones no requieren el procesamiento adicional de una llamada a una función normal. Normalmente la definición de funciones en línea van precedidas de la palabra **inline** pero, aunque la declaración de la función no esté precedida de la palabra clave **inline**, cualquier función definida en el interior de la declaración de una clase es creada automáticamente en línea.

```
class Telefono {
  string numero;
  string tipo_tarifa;
  string usuario;
  float importe;
  void inicializarImporte () { importe=0; }
public:
  float obtenerImporte( );
};
```

8.5. Resumen

El lenguaje C++ es mucho más que un lenguaje que soporta el paradigma de programación orientada a objetos. Como ya se ha comentado a lo largo de este libro, gran parte de su éxito se basa en su carácter multiparadigma. Este lenguaje engloba conceptos de programación estructurada, programación orientada a objetos y programación genérica.

Por otra parte, su librería estándar es uno de sus principales valores. En este capítulo, se ha tratado de mostrar una ligera pincelada de su alcance. Sin embargo, para dominar este lenguaje de programación es necesario conocer con mayor profundidad esta librería.

cc BY NC SA

Apéndice A

Errores y ejercicios

A.1. Relación de los errores más comunes

A continuación se citan algunos de los errores más comunes producidos por los programadores cuando comienzan a familiarizarse con la programación orientada a objetos con el lenguaje C++. El objetivo de este apartado es que el lector conozca tales errores para no cometerlos cuando programe en este lenguaje.

- Acceder a atributos miembro privados. El acceso directo a los atributos miembro privados de una clase sólo está permitido para las funciones miembro de la misma clase.
- Constructores y destructores. Cuando no se crea un constructor o destructor para una clase o se crean éstos como privados, debe ser el programador el que controle la inicialización o destrucción de los valores de los atributos pertenecientes a los objetos que se creen de una clase.
- Creación de objetos de forma dinámica. Cuando se crea un objeto de forma dinámica, es decir cuando el programador crea objetos como apuntadores a una clase, debe ser el mismo programador quien deba eliminar el objeto creado dinámicamente de una forma explícita (utilizando como se ha visto el operador **delete**).
- Utilización del puntero **this**. Si no se utiliza el puntero **this** en funciones que reciben como parámetros variables que tengan el mismo nombre que los atributos privados de una clase, no se actualizarán de forma correcta dichos atributos, produciéndose una ejecución errónea de dichas funciones miembro.
- Paso de objetos como parámetros por valor o retorno de objetos por valor. Cuando se pasa un objeto como parámetro por valor a una función, hay

que ser consciente de que el compilador creará una copia del mismo, invocando al constructor copia. Además, cuando se llegue al final del cuerpo de la función, se destruirá esa copia, invocando su destructor. Por lo tanto, se debe definir el constructor copia de los objetos que vayamos a pasar como parámetros por valor.

- Heredar funciones y atributos privados. Los miembros privados de una clase sólo son accesibles desde dentro de la misma. Por lo tanto, intentar acceder a ellos directamente desde una clase derivada dará error.
- Redefinir atributos o funciones miembro por error. Cuando se usa el mecanismo de herencia para crear una clase, es necesario ser consciente de los atributos o funciones miembro de la clase base para redefinir o sobrescribir sólo los adecuados. Si no se tiene cuidado, se pueden redefinir miembros de la clase base por error e introducir errores en el código.
- Crear objetos de clases abstractas. Por definición no se puede crear una instancia de una clase abstracta. El comportamiento recomendado consiste en crear una clase derivada de la clase abstracta que implemente todas las funciones miembro. Y crear objetos de esta nueva clase.
- No definir todos las funciones virtuales puras de una clase abstracta. Si crea una clase a partir de una clase abstracta y se quiere que sea instanciable, hay que asegurarse de proporcionar un cuerpo para todas las funciones virtuales puras. De lo contrario, la nueva clase seguirá siendo abstracta.
- Olvidar declarar el destructor de la clase raíz de una jerarquía como virtual. Si no lo hacemos, cuando se elimine un objeto de una clase derivada, no se producirá la secuencia de invocaciones de los destructores correctamente, produciendo posibles errores de no liberación de recursos como, por ejemplo, pérdida de memoria dinámica.
- Usar la estructura de control **switch** en vez de hacer un uso efectivo del polimorfismo. El polimorfismo nos permite obtener un comportamiento diferente para cada tipo de objeto sin tener que añadir complejidad a las funciones. Las funciones que trabajan con objetos de la clase base no necesitan ser modificadas para poder trabajar con objetos de clases derivadas. Por lo tanto, no hace falta incluir una estructura de control **switch** en dichas funciones para obtener un comportamiento distinto para cada tipo de objeto.

A.2. Ejercicios

En este apartado se propone una serie de problemas que el alumno debe intentar resolver para afianzar los conceptos explicados dentro de este libro.

A.2.1. Clases y objetos

1. Definir la clase *Alumno* que contenga los datos relativos a un alumno de universidad que el lector crea necesarios como nombre, dirección, ciudad, especialidad, número de asignaturas, etc. Creando las funciones miembro constructor predeterminado y destructor de la clase.

2. Crear tres funciones miembros que realicen lo siguiente:

   - *calcular_ media*. Esta función deberá calcular la nota media del alumno devolviéndola como salida
   - *devolver_ especialidad*. Esta función devolverá la especialidad de la que esté matriculado un alumno
   - *mostrar_ alumno*. Esta función mostrará por pantalla los datos relativos a un alumno

3. Definir un nuevo constructor parametrizado para la clase *Alumno* que reciba como parámetros de entrada la *especialidad*, la *ciudad* y el *número de asignaturas* con los valores siguientes:

   - Especialidad: "IT Informática"
   - Ciudad: "Cáceres"
   - Nº de Asignaturas: 5

   que inicialice las variables miembro de los nuevos objetos con estos valores por defecto.

4. Definir un constructor copia para la clase *Alumno*.

5. Realizar las siguientes operaciones en el programa principal:

   - Crear tres objetos de la clase *Alumno* utilizando el constructor predeterminado, el constructor parametrizado y el constructor copia respectivamente.
   - Mostrar por pantalla el contenido de los tres objetos creados anteriormente utilizando la función miembro *mostrar_ alumno*.
   - Añadir un mensaje en el destructor de la clase *Alumno* y comprobar para los 3 objetos anteriores si se destruyen de forma correcta ambos objetos.

6. Crear una clase universidad que posea una única función miembro estática llamada *examen*. Esta función debe recibir dos parámetros: una cadena con el nombre de una asignatura y un alumno. La función examen debe asignarle una nota para esa asignatura al alumno que se pasa por parámetro (construir las funciones miembro de alumno necesarias). En la

función main, crear un alumno, asignarle unos valores iniciales e invocar a la función examen pasándole este objeto alumno por parámetro. Probar a construir la función examen recibiendo el parámetro alumno por valor y por referencia. Hacer un análisis de las llamadas a constructores y destructores de la clase alumno, razonando por qué ocurren.

A.2.2. Mecanismos de reutilización

1. Crear una clase derivada de la clase *Alumno* denominada *AlumnoIntercampus*, incluyendo un atributo miembro para recoger el pais de origen del alumno y redefiniendo las funciones de la clase base que sea necesario. Se deben implementar el constructor predeterminado, constructor copia, un constructor con parámetros y el destructor.

2. A partir de la siguiente descripción de objetos, crear la mejor jerarquía de clases posible.

   - Alumno de Informática
     a) Atributos: nombre, edad, curso
     b) Acciones: levantarse, ir a clase, examinarse, programar, comer, dormir
   - Alumno de Arquitectura
     a) Atributos: nombre, edad, curso
     b) Acciones: levantarse, ir a clase, examinarse, dibujar, comer, dormir
   - Alumno de Obras Públicas
     a) Atributos: nombre, edad, curso
     b) Acciones: levantarse, ir a clase, examinarse, diseñar, comer, dormir
   - Profesor
     a) Atributos: nombre, edad, especialidad, materias
     b) Acciones: levantarse, dar a clase, evaluar, investigar, comer, dormir

3. Crear una clase *Escuela* que contenga un array de 10 alumnos. Incluir en ambas clases la definición del constructor predeterminado, constructor copia y el destructor. Se recomienda también que cada constructor y destructor muestre un mensaje significativo por pantalla. Una vez implementada la clase *Escuela*, incluir el siguiente código en el programa y analizar las llamadas que se producen a constructores y destructores explicando su motivo:

```
void inspeccion(Escuela *esc){
  cout<<''Inspección\n'';
}
void fiesta(const Escuela& esc){
   cout<<''Fiesta\n'';
}
int main(){
  Escuela escuela;
  inspeccion(escuela);
  fiesta(escuela);
  cout<<''Fin del programa\n'';
}
```

A.2.3. Polimorfismo

1. Convierte la jerarquía de clases del ejercicio 2 del apartado A.2.2 a una jerarquía polimórfica. Crea un array polimórfico que almacene elementos de esa jerarquía y una función que reciba un argumento del tipo base de esta jerarquía e invoque a las funciones definidas en el interfaz base.

2. Comprueba que se producen la secuencia correcta de llamadas a constructores y destructores cuando se crea y destruye un objeto de la clase *AlumnoInformatica*.

3. Haz que la clase base de la jerarquía anterior sea una clase abstracta.

A.2.4. Excepciones

1. Realizar la función *leer_ edad* que lea la edad de una persona desde línea de comando y lance una excepción si la edad es de tipo cadena o bien si esta edad es menor que cero o mayor de 120.

2. Realizar un programa que lance una excepción a una función y dentro de esta función se relance otra excepción al programa principal.

3. Crear una clase *Coche* que contenga las variables miembro que se indican y que en su constructor lance una excepción si no puede reservar memoria para la variable miembro *marca* y en su destructor lance una excepción si no ha sido reservada memoria para la variable miembro *pais_ origen*.

```
class Coche {
  char *marca;
  char *fecha_fabricacion;
  char *pais_origen;
};
```

A.2.5. Identificación y modelado

1. Una universidad desea que la gestión de la contratación de nuevos profesores se haga a través de una aplicación que funcione del siguiente modo: La universidad está formada por diversos departamentos (Informática, Matemáticas, Física, etc.), los cuales, cuando necesiten cubrir una vacante, enviarán una solicitud al sistema con los detalles de la plaza a cubrir. El sistema debe publicar entonces la oferta para cubrir esas plazas vacantes con el plazo de tiempo disponible para responder, los requisitos mínimos que deben cumplir los candidatos y las condiciones del contrato. Durante el tiempo de vigencia de la oferta, el sistema ofrece un formulario de méritos que los posibles candidatos deben rellenar. Una vez terminado el plazo, el sistema elabora una lista con los candidatos ordenados en función de sus méritos. El sistema toma entonces al primero de la lista para comunicarle que ha ganado la plaza. Si este candidato renuncia a ella, el sistema se lo comunica al siguiente candidato, y así sucesivamente hasta encontrar un candidato que acepte. Cuando el candidato firma el contrato, el sistema debe mantener toda la información relativa al nuevo profesor dentro del sistema. Hay que señalar que la Universidad no desea mantener información de los candidatos que no son contratados una vez que la vacante ha sido cubierta y que los profesores de la Universidad no pueden pertenecer a más de un departamento. Se pide realizar el diagrama de casos de uso y de clases del sistema propuesto. También se pide elaborar el diagrama de secuencias de cada caso de uso obtenido.

cc BY NC SA

Bibliografía

[1] Booch, G, Rumbaugh, J., y Jacobson, I.: *The Unified Modeling Language User Guide* (1ª ed.). Addison-Wesley, 1998.

[2] Bruegge, B. y Dutoit, Allen,A.H.: *Object-Oriented Software Engineering* (1ªed). Prentice-Hall, 2000.

[3] Eckel, B.: *Thinking in C++, Volume 1: Introduction to Standard C++* (2ª ed). Prentice Hall, 2000.

[4] Eckel, B. *Thinking in C++, Volume 2: Practical Programming* (2ª ed). Prentice Hall , 2003.

[5] Joyanes Aguilar, L.: *Fundamentos de programación. Algoritmos, estructuras de datos y objetos*. McGrawHill, 2003 (3ª Edición).

[6] Meyer, B.: *Construcción de Software Orientado a Objetos*. Prentice Hall, 2000.

[7] Pressman, R.S., adapted by Ince D.: *Software Engineering. A Practitioner's Approach. European Adpatation* (5ª ed). McGraw-Hill, 2000.

[8] Savitch W.: *Resolución de problemas con C++* (3ª ed). Prentice Hall, 2000.

[9] Sommerville, I.: *Software Engineering* (6ª ed). Pearson Education Limited, 2001.

[10] Stroutrup, B.: *El Lenguaje de Programación C++*. Addison Wesley Publishing Company, 1998.

Programación Orientada a Objetos

La Programación Orientada a Objetos se ha convertido en el paradigma de programación más extendido en la actualidad. Sus conceptos y técnicas son ampliamente utilizados para reducir la complejidad del desarrollo de grandes sistemas software.
Este paradigma tuvo su origen dentro del mundo académico universitario y sigue evolucionando gracias a esfuerzos de investigación tanto de ámbito universitario como empresarial.
Este libro nace con la ambición de recopilar los conceptos básicos de este paradigma y exponerlos de una forma sencilla y práctica.