Programación orientada a objetos

C++

Sergio Rogelio Tinoco Martínez
Armando Molina Moreno
Francisco Cortés Arredondo

SEP

conalep Michoacán

Semestre 4

PRESENTA:

Programación orientada a objetos

Autores

Sergio Rogelio Tinoco Martínez
Armando Molina Moreno
Francisco Cortés Arredondo

Coordinadores:
Lic. José Azahir Gutiérrez Hernández
Ing. Eduardo Ochoa Hernández
Lic. Filho Enrique Borjas García

Título original de la obra:

Programación orientada a objetos. Copyright © 2013-2014 por CONALEP/CIE. Gral. Nicolás Bravo No. 144, Col. Chapultepec norte C.P. 58260, Morelia Michoacán, México. Tel/fax: (443) 113-6100 Email: arturo.villasenor@mich.conalep.edu.mx

Registro: **CONALEP-LIB-C++-01A**

Programa: Profesor escritor, objetos de aprendizaje. Desarrollo de la competencia de la producción de información literaria y lectura.

Esta obra fue publicada originalmente en Internet bajo la categoría de contenido abierto sobre la URL: http://www.cie.umich.mx/conalepweb2013/ mismo título y versión de contenido digital. Este es un trabajo de autoría publicado sobre Internet Copyright © 2013-2014 por CONALEP Michoacán y CIE, protegido por las leyes de derechos de propiedad de los Estados Unidos Mexicanos. No puede ser reproducido, copiado, publicado, prestado a otras personas o entidades sin el permiso explícito por escrito del CONALEP/CIE o por los Autores.

Tinoco Martínez, Sergio R. *et al.* (2014) **Programación orientada a objetos.**
México: CONALEP/CIE

xiv, 185 p.; carta

Registro: **CONALEP-LIB-C++-01A** Documentos en línea

Editores:
Ing. Eduardo Ochoa Hernández
Lic. Filho Enrique Borjas García

Quedan rigurosamente prohibidas, sin la autorización escrita de los titulares del "Copyright", bajo las sanciones establecidas por la ley, la reproducción total o parcial de esta obra por cualquier medio o procedimiento, comprendidos la reprografía y el tratamiento informático, y la distribución de ejemplares de ella mediante alquiler o préstamo público.

©2013-2014 Morelia, Michoacán. México
Editorial: CONALEP Michoacán
Col. Chapultepec norte, Gral. Nicolás Bravo No. 144, Morelia, Michoacán.
http://www.cie.umich.mx/conalepweb2013/

Registro: **CONALEP-LIB-C++-01A**
ISBN: En trámite
Impreso en_____
Impreso en México –Printed in Mexico

DIRECTORIO

Lic. Fausto Vallejo Figueroa
Gobernador Constitucional del Estado de Michoacán

Lic. J. Jesús Sierra Arias
Secretario de Educación

Mtro. Álvaro Estrada Maldonado
Subsecretario de Educación Media Superior y Superior

Ing. Fernando Castillo Ávila
Director de Educación Media Superior

M.A. Candita Victoria Gil Jiménez
Directora General del Sistema CONALEP

M.C. Víctor Manuel Lagunas Ramírez
Titular de la Oficina de Servicios Federales en Apoyo a la Educación en Michoacán

Dr. Salvador Jara Guerrero
Rector de la Universidad Michoacana de San Nicolás de Hidalgo

Lic. José Arturo Villaseñor Gómez
Director General del CONALEP Michoacán

Lic. José Azahir Gutiérrez Hernández
Director Académico

L.E. Rogelio René Hernández Téllez
Director de Planeación, Programación y Presupuesto

Lic. Faradeh Velasco Rauda
Directora de Promoción y Vinculación

Ing. Mónica Leticia Zamudio Godínez
Directora de Informática

Lic. Víctor Manuel Gómez Delgado
Director de Servicios Administrativos

Ing. Genaro González Sánchez
Secretario General del SUTACONALEPMICH

Tec. Juan Pineda Calderón
Secretario General del SUTCONALEP

Prefacio

Estimado estudiante:

Las palabras que sombrean estas páginas, no son simple ciencia dentro del diálogo como depósitos de datos e información, ni son cuestión de vocabulario o listado de definiciones, son la experiencia generosa de la comunidad CONALEP Michoacán, esa realidad oculta pero necesaria que respaldó las tareas de investigación y composición literaria del discurso que integra este libro. Nos referimos a los profesores, administrativos y sindicatos que hoy convergen en el umbral de la existencia para apoyar a un grupo de profesores escritores que han creado en el sereno libre, arquitecturas de conocimientos como un viaje de aprendizaje que exigirá del estudiante, lo mejor de sí mismo ante la presencia luminosa del texto, ese que pretende enseñarle a caminar con la frente en alto.

Las ideas asociadas en este libro, equivalen a la imaginación lograda en el acto de escribir desde otros textos, al decodificarlas el estudiante, se le exige más vocabulario para enriquecer su habla y hacer ver a sus ojos más allá de la estrechez de la información que inunda a la sociedad moderna. El libro no presenta la superficie de la existencia como cruda observación, procura que su dificultad incite a perforar la realidad hasta reflexiones que renueven los modos inciertos de dar significado al mundo. La ciencia, la literatura y la tecnología no las percibimos como mundos incomunicables, los valores son explícitos caminos que las vinculan entorno al currículo del técnico bachiller. Tienen estos textos organización de premisas, técnicas, justificaciones, normas, criterios y como Usted se dará cuenta, también mostrará nuestros límites para seguir haciendo puentes entre las incesantes creaciones de nuevas fronteras de la investigación científica y técnica. Se pretende que estos libros sean contenido y no un libro de prácticas escolares, sean la herramienta de complementación para enriquecer los discursos de la enseñanza-aprendizaje.

Los profesores de CONALEP enfrentan a diario las carencias visibles de medios tecnológicos, materiales y documentales, sería fácil usar las palabras para señalar hasta el cansancio nuestras apremias, pero se ha decidido mejor producir libros como testimonios vivos y luminosos que renueven el rol social de la academia colegiada sensible a la condición social, susceptible de ir perfeccionándose con la acumulación de esta experiencia literaria, para servir de mejor manera al enriquecimiento de las competencias necesarias para realizar el sueño de éxito de tantos jóvenes Michoacanos.

Lic. José Arturo Villaseñor Gómez
Director General del CONALEP Michoacán

Mensaje a la comunidad académica

Se vive una época difícil para los libros, no son momentos de lectura tampoco, sin embargo, esto no debe ser un motivo para no producir literatura para educar mentes creativas, nuestra generación considera importante que sea la literatura la que emocione a realizar los grandes sueños de nuestros jóvenes. Si el libro escrito por mexicanos para mexicanos se perdiera, la conciencia de nuestra sociedad quedaría en orfandad, congelada de sentido y seríamos habitantes de un mundo siempre detrás de máscaras en rituales de simulación. Este argumento presente en *El laberinto de la soledad*, Octavio Paz nos dice al hombre moderno, *seamos universales sin dejar de ser diferentes*. El efecto de producir literatura en una sociedad, es como dice Steven Pinker, es "sacar los ángeles que llevamos dentro", es lo mejor de nuestro ser como oferta de conocimiento y valores al servicio de nuestra sociedad. El efecto Malinche que prefiere libros de traducción, ocasiona la pérdida de identidad y crear la cultura de producir experiencias de conocimiento como herencia educativa para nuestros estudiantes. Los aztecas en su derrota se sintieron abandonados por sus dioses, en el presente si cancelamos que la voz de nuestros profesores que hablarán desde libros a las nuevas generaciones, es algo equivalente, al negar que las ideas en su crecimiento son una respuesta sociolingüística económica y democrática de identidad de una nación. Michoacán es un proyecto histórico de libertad, sin ignorar la realidad del malestar de la cultura actual, CONALEP desarrolla un programa académico para impulsar su capacidad y compromiso social para generar las ideas curriculares para enriquecer la sensibilidad y la imaginación científica, técnica y humanista de su comunidad.

Producir literatura curricular en CONALEP Michoacán, es asumir su personalidad moral, como dueña de una conciencia movida para una educación con la pasión de razones y expresiones culturales con la competencia para transformar positivamente la realidad. Aún cuando esta realidad adversa, de actitudes de autoridades renuentes a transformar la realidad educativa y de presiones económicas, hoy entregamos esta literatura escrita con profesores de CONALEP, máxime reconocimiento, porque trabajaron de manera altruista durante extenuantes y largas jornadas de trabajo; la comunidad CONALEP y la sociedad Michoacana toda, reciban esta obra como testimonio de la grandeza de este histórico Michoacán.

Lic. José Azahir Gutiérrez Hernández
Director Académico

Enfoque por competencias

"El hombre sigue siendo la computadora más extraordinaria de todas".
John Fitzgerald Kennedy

La informática se ha convertido en el área omnipresente por antonomasia. Comenzando con los grandes centros de cómputo y llegando hasta los teléfonos celulares miniatura, que incluso niños pequeños ya consideran parte de su ser, las computadoras ya no son parte de nuestras vidas, *son* nuestras vidas. Imaginar hoy una vida sin computadoras es prácticamente imposible. Su importancia es tal, que quienes dominan esta herramienta a escala global son los que marcan el paso de la evolución humana.

La satisfacción que proporciona el poder dominar a una computadora es inmensa aunque no apta para todo ser humano. Existe una raza muy especial entre la humanidad a la que se le ha concedido el nombre de *programadores* o, más recientemente, *desarrolladores*. Este grupo de mujeres y hombres *evoluciona* cada determinado tiempo una forma nueva de pensar o *paradigma*, la cual se transmite en última instancia hacia *la herramienta*, la computadora.

Considerando lo anterior, el programador que enfrenta la tarea difícil de proporcionarle inteligencia a *"un tonto muy rápido"*, la computadora, ve facilitada su tarea con la aplicación del paradigma orientado a objetos. La competencia en la programación orientada a objetos (*POO*) se fundamenta en el dominio de los conceptos *abstracción*, *encapsulación*, *herencia* y *polimorfismo*. Partiendo de ahí, el técnico bachiller podrá modelar más fácilmente el mundo *real* por medio de la computadora, y resolver problemas de mayor envergadura que con el paradigma estructurado clásico.

No puede faltar, por supuesto, la expresión de la POO en un lenguaje con el cual el desarrollador de software se comunique con la computadora. En este sentido, en esta obra se eligió el lenguaje C++, por ser la eficiencia el objetivo principal de su desarrollo.

En toda obra literaria se afirma una realidad independiente de la lengua y del estilo: la escritura considerada como la relación que establece el escritor con la sociedad, el lenguaje literario transformado por su destino social. Esta tercera dimensión de la forma tiene una historia que sigue paso a paso el desgarramiento de la conciencia: de la escritura transparente de los clásicos a la cada vez más perturbadora del siglo XIX, para llegar a la escritura neutra de nuestros días.

Roland Barthes, *El grado cero de la escritura*

Palabra escrita bajo luz

En un mundo cada día con más canales de comunicación, la palabra escrita camina por los muros que denuncian el drama catastrófico sobre el medio ambiente y sobre el control de la vida humana; el combustible de esta desesperanza produce apatía profunda por tener contacto con el mundo de la literatura, esta distorsión moral parece reflejarse entre los que no quieren sentir responsabilidad ni pensar, dejando a otros su indiferencia al ser prisioneros de ligeras razones y tirria justificada en la empresa de sobrevivir.

Especular en un mundo sin libros es exponer al mundo a la ausencia de pensamiento, creatividad y esperanza. Los libros, dedicados a ser arrebatados por el lector, están expuestos a ser poseídos por las bibliotecas vacías y que con el tiempo opacan sus páginas y empolvan la cubierta de lo que alguna vez fue un objeto de inspiración. Resulta difícil transcribir este instante de un peligroso espacio donde ya muy pocas palabras sobreviven dentro de la reflexión y las pocas sobrevivientes han abandonado la unión del sentido de vivir y el sentido del pensar científico. Escribir pasa de ser un placer repentino a ser una necesidad inminente, es el puente entre lo conocido y lo inexplorado. Es un reto de hoy en día inmiscuirse en lo que una vez fue lo cercano y dejar de lado la novedad tecnológica para poder a través de las barreras que nos ciegan abrir fronteras literarias. Es un proyecto que conspira a favor de la libertad creativa, de la felicidad lúcida cargada de libros embajadores de nuevas realidades.

Entre un mar de razones dentro del libro escolar en crisis, se percibe la ausencia de esa narrativa del cuerpo del texto, misma que alimenta al lector de una experiencia de conocimiento, su ausencia, es más un mal glosario, de un mal armado viaje literario científico o de ficción. En esos viajes de libros en crisis, nos cansamos de mirar espacios vacíos de talento, emociones y sensibilidad para responder a un entorno adverso; son muchas veces un triunfalismo de autoevaluación y una falsa puerta de una real competencia para actuar en la realidad. Uno no solo vive, escucha la voz interior de un libro, uno es fundado en el manejo del lenguaje que explica, crea, aplica o expande los límites del horizonte de nuestro imaginario actuante en lo real. No vivimos leyendo texto, sino leyendo el paisaje de una realidad, el libro toma la voz del progreso en una siempre reconstrucción lingüística del sujeto que explica, transforma y comunica desde los desafíos de su generación.

La información cruda que tanto rellena los libros grises, oscuros y papel pintado; requiere ser dotada de conceptos que permitan alimentar al sujeto que toma decisiones, que explora con paso lento, que mira por dentro del lenguaje y aplica la información que cobra sentido en la siempre expansión de las ideas.

Escribir un libro es siempre reconstruir un discurso, sus lectores en este discurso son el puente a un texto profundo que demanda esfuerzo en la reconstrucción de los procesos de razonamiento y el entretejido del discurso que involucra información de fondo, esas fuentes que justifican su análisis y poseen significado privilegiado para la comprensión de una realidad.

El lector puede hacer uso del libro con su propia experiencia y con su autoayuda, al precisar términos y conceptos para prolongar su horizonte de interpretación, el libro se hace cargo de

la memoria de un plan de estudios, es un discurso de diferentes capas de argumentos, tras este texto se anuncia un orden de experiencia propuesto para su aprendizaje. El libro está conformado para jóvenes con memoria sin dolor para nuevas palabras, aborda el olvido como una deficiencia de interactividad entre el discurso argumentativo y los referentes conceptuales. Esto es el reto en la producción de los libros CONALEP. La propuesta es una reconstrucción de una semántica más profunda, como el principal reto del estudiante técnico bachiller del siglo XXI.

Libro,…

todos te miran,

nosotros te vemos bajo la piel.

SUMARIO

Segunda parte
Funciones

Tercera parte
Introducción a la Programación Orientada a Objetos

Cuarta parte
Implementación de la POO en C++

Capítulo 0. Introducción

"… Si la programación es como las matemáticas ¿por qué los programas siempre tienen errores?

Las matemáticas están bien o están mal, el software tiene errores…

… Si los programadores son como (el resto de) los ingenieros, yo debería poder cambiar un programador por otro y obtener resultados similares ¿no?

Nosotros (los programadores) somos como escritores de novelas… el arte de la escritura es un proceso raro, insondable y artístico que no puede garantizar los resultados…"

Writing Software is Like… Writing
Bruce Eckel

La presente obra está dirigida al estudiante CONALEP (sin restringirse de manera exclusiva a él) que ya ha iniciado su camino en el arte y la ciencia de la programación. Se presupone, especialmente, que ya ha tenido contacto directo con la terminología y conceptos del diseño estructurado de algoritmos. No obstante lo anterior, se revisará brevemente el ciclo de desarrollo de software[1], es decir, el proceso general mediante el cual se escribe un programa para indicarle a un sistema de cómputo las tareas que el programador desea que este realice. Así pues, se explorará la etapa del entendimiento del problema; se describirá y desarrollará un algoritmo para resolverlo (empleando, para su expresión, la notación del pseudocódigo y/o de los diagramas de flujo[2]) y, con entradas sencillas de ejemplo, se pondrá a prueba; la solución determinada se codificará, además, en un lenguaje de programación (en esta obra, en el lenguaje de programación C++); y, finalmente, el programa desarrollado se compilará y probará en un equipo de cómputo.

Como su nombre lo indica, este proceso es un *ciclo*, el cual se repetirá tantas veces como sea necesario, a fin de resolver el problema planteado inicialmente.

La parte medular de esta obra consiste en construir un puente sólido y sencillo, el cual facilite el paso entre el paradigma estructurado, que a estas instancias ya debe dominar el lector, y el paradigma de la programación orientada a objetos. Esto no es algo simple, como no lo es cualquier cambio de paradigma, y demanda un gran esfuerzo por parte del lector, a fin de que derribe los *muros abstractos* que psicológicamente crea, al sentirse cómodo en un paradigma específico (el estructurado, en este caso).

El conocimiento del paradigma estructurado, representado para un estudiante CONALEP por los módulos correspondientes al núcleo de formación básica: *Manejo de técnicas de programación* (diseño estructurado de algoritmos) y *Programación básica* (lenguaje de programación C), o sus equivalentes en los diferentes mapas curriculares; sirve como fundamento para el tránsito hacia el paradigma orientado a

objetos. Esto es, en el presente trabajo transformamos la algoritmia, y su concretización en el lenguaje de programación C, en un enfoque orientado a objetos, expresado a través del lenguaje de programación C++, recurriendo también a la parte imperativa/estructurada de este último.[i]

Con relación al paradigma orientado a objetos, se hace primeramente un tratamiento de los conceptos fundamentales de esta tecnología, dado que es de importancia vital su comprensión, de forma *previa* a la exposición de su implementación en C++ o en el propio *Lenguaje Unificado de Modelado* (*UML – Unified Modeling Language*).[3] El aprendizaje del UML antes de los conceptos de la tecnología orientada a objetos es similar a aprender a leer un diagrama eléctrico, sin conocer primeramente los conceptos fundamentales de la electricidad[4].

Expuesta la teoría general de objetos, se muestra cómo concretizarla con su implementación en el lenguaje C++. Cabe señalar, por tanto, que este libro no es una exposición exhaustiva ni del paradigma estructurado, ni del orientado a objetos, más bien, es una introducción *suficiente* para que el lector se interese en el tema, tenga los fundamentos del mismo, y esté en posibilidades de emprender el estudio de obras más avanzadas que complementen su enseñanza.

[1] S. Horstmann, Cay. (**2010**) C++ for Everyone, 2nd. Ed. New Jersey: John Wiley & Sons, Inc.

[2] A. Robertson, Lesley (**2004**) Simple Program Design: A Step-by-Step Approach, 4th. Ed. Hong Kong: Course Technology.

[3] Fowler, M. & Scott, K. (**1999**) UML gota a gota. Naucalpan de Juárez, México: Addison-Wesley.

[i] No obstante, la revisión que se realiza del paradigma estructurado no es exhaustiva, es decir, conceptos como variable, operador, expresión, estructuras de control (condicionales o iterativas), diagramas de flujo, pseudocódigo, por mencionar algunos, no se analizan en profundidad. Como antes se menciona, en esta obra se da por hecho que el lector domina tales conocimientos.

[4] A. Weisfeld, Matt. (**2009**) The Object-Oriented Thought Process, 3rd. Ed. New Jersey: Addison-Wesley.

Capítulo 1. La herencia del paradigma estructurado o *el alma del lenguaje de programación C++*

"… No puedo dar instrucciones completas aquí sobre cómo aprender a programar –es una habilidad compleja. Pero puedo decir que los libros y los cursos no lo harán –muchos, quizás la *mayoría* de los mejores hackers (en el mundo) son autodidactas. Puedes aprender las características de los lenguajes –pedacitos de conocimiento– de los libros, pero la mentalidad que transforma ese conocimiento en habilidades innatas, sólo se puede aprender mediante la práctica y el aprendizaje. Lo que lo conseguirá es (a) *leer código* y (b) *escribir código*…"

How To Become a Hacker
(Cómo convertirse en un hacker)
Eric Steven Raymond

Como ya se indicó en la introducción, esta obra trata principalmente sobre la programación orientada a objetos (*POO*), pero se revisa la implementación del paradigma estructurado (en el lenguaje C++) como antecedente y fundamento de la misma.

1.1. Historia breve del lenguaje C++

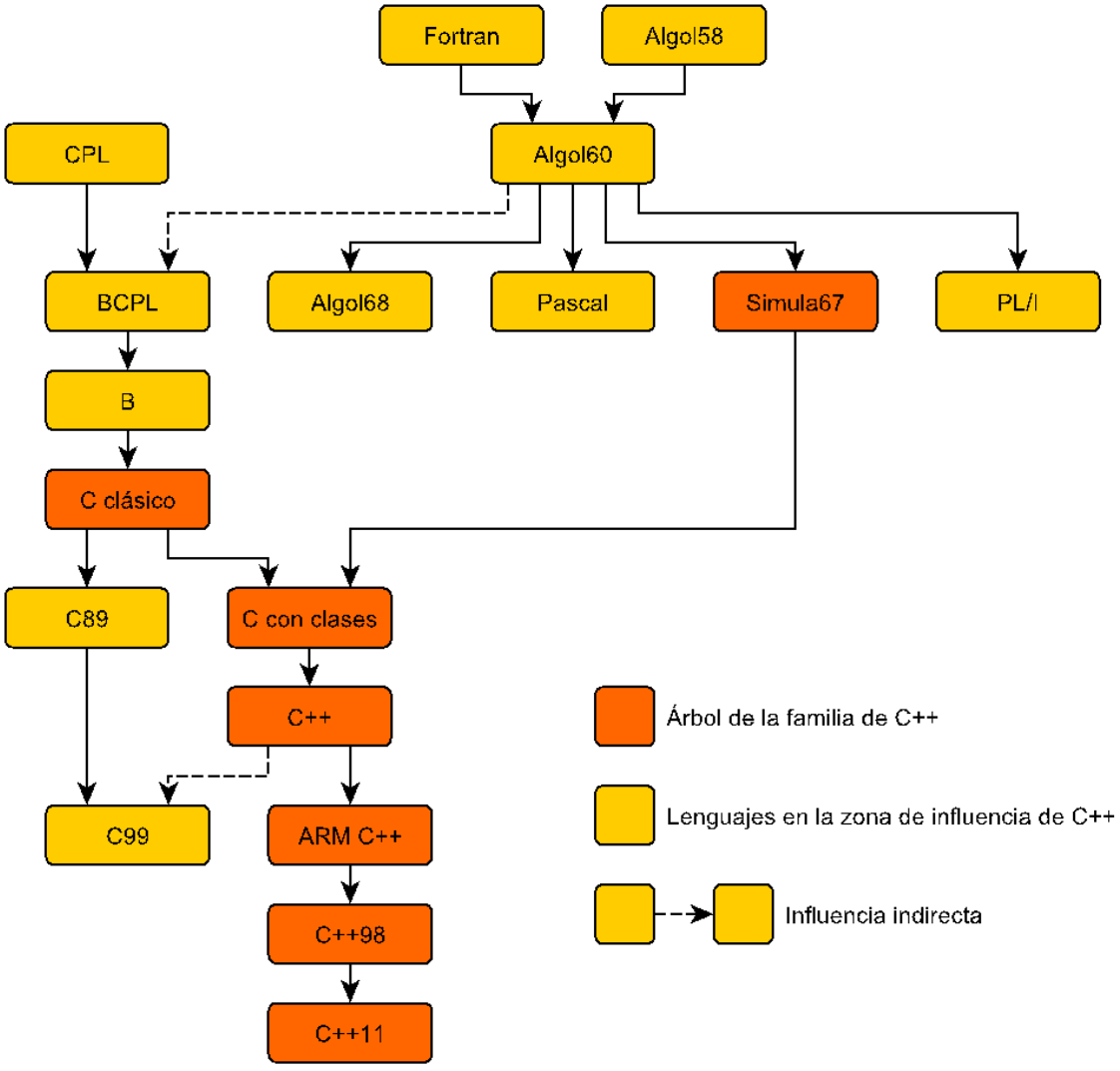

Figura 1. Genealogía del lenguaje de programación C++

La Figura 1 presenta el árbol genealógico del lenguaje de programación C++. El desarrollo de este lenguaje no puede ser mejor relatado que por su propio creador: Bjarne Stroustrup.[1]

Stroustrup señala que la primera influencia de C++ proviene del lenguaje Fortran (*FORmula TRANslation*), desarrollado inicialmente en 1956 y considerado el primer lenguaje de programación de alto nivel, es decir, el primer paso dado hacia la implementación de los programas en el *dominio del problema*.[i,2] Fortran ya cuenta con arreglos, ciclos de repetición, una librería de funciones matemáticas, mecanismos de entrada/salida, funciones definidas por el usuario y, entre las cuestiones más importantes logradas con su desarrollo, se descubrió la estructura básica de un compilador así como la notación *Backus-Naur Form* (*BNF*) para expresar la gramática de un lenguaje de programación.[ii]

El primero de los lenguajes de programación que empleó la notación BNF, fue Algol60. El *lenguaje algorítmico* (***ALGO****rithmic* ***L****anguage*), Algol, fue desarrollado por un grupo de gente auspiciada por la Federación Internacional de Procesamiento de Información (*IFIP – International Federation of Information Processing*), el cual estableció los fundamentos para los lenguajes de programación modernos, incluido C++, con las características implementadas en Algol: tipos de datos en tiempo de compilación, ámbito léxico, uso de una gramática para definir el propio lenguaje, separación de las reglas sintácticas de las semánticas así como de la definición del lenguaje con respecto a su implementación y, al final pero no menos importante, el soporte para la programación estructurada. Aunque Algol no tuvo gran éxito, por ser *demasiado diferente* a lo que existía, fue un parteaguas que estableció las bases para el futuro de la programación.

[i] La programación se modela de una forma más cercana al problema del mundo real que se intenta resolver, por lo que es más fácil de entender para los programadores y, por lo tanto, se aleja más del modelo entendible por las computadoras.

[ii] La notación BNF es empleada prácticamente por todos los lenguajes de programación modernos, a fin de expresar su gramática y para la construcción de sus compiladores y/o intérpretes.

Siguiendo el linaje de Algol, a mediados de la década de los 60's se desarrolló el lenguaje Simula, como un superconjunto de Algol60 y como el primer lenguaje de programación en implementar los conceptos que hoy definen a la POO: la encapsulación, la herencia y el polimorfismo.[iii] Simula acuñó los términos *clase*, para denominar a un tipo definido por el usuario; y *virtual*, para denotar una función que puede ser sobrescrita e invocada a través de una clase *base* o *padre*. Así, un programa se convierte en un conjunto de objetos interactuando entre ellos y no en un monolito de código.[iv]

A través de otra derivación de Algol, diferente a la que originó Simula, llega la principal influencia del lenguaje C++, el lenguaje de programación C, conocido también como su lenguaje *padre*, por la cantidad tan grande de características que heredó de él.

El origen de C proviene del proyecto CPL que nunca se completó en Inglaterra; del lenguaje BCPL (CPL Básico – *Basic CPL*) que Martin Richards creó cuando visitó el Instituto Tecnológico de Massachusetts, en Estados Unidos, proveniente de la Universidad de Cambridge; y de un lenguaje interpretado denominado B, creado por Ken Thompson.

CPL fue un proyecto conjunto entre la Universidad de Cambridge y el Colegio Imperial en Londres. Como el proyecto se inició en Cambridge, el significado del nombre, "*C*", correspondía oficialmente a "*Cambridge*". Luego entró al proyecto el Colegio Imperial y el significado cambió a "*Combinado*". Se supone que en realidad su significado siempre fue "*Christopher*", derivado del nombre del diseñador principal de CPL, Christopher Strachey.

[iii] Aún y cuando Simula ya emplea estos conceptos, no lo hace propiamente hablando con los términos *encapsulación* y *polimorfismo*, pero sí *herencia*.

[iv] La forma monolítica del software a la que se hace referencia, se debe colocar en el contexto apropiado de tiempo, el del nacimiento del lenguaje Simula, ya que esta es una de las modificaciones que la programación estructurada introdujo, que el software se desarrollara en módulos de tamaño *suficientemente* manejable y no como tales monolitos.

Dennis Ritchie diseñó e implementó el lenguaje de programación C en el Centro de Investigación en Ciencias Computacionales de los Laboratorios Bell, en Murray Hill, Nueva Jersey. Su trabajo fue un esfuerzo realizado como una clara contraposición a la costumbre establecida en esa época de que el *software de sistemas*[v] debía programarse en lenguaje ensamblador, generando la imposibilidad de ser portable. Por tal motivo, Ken Thompson y el propio Dennis Ritchie desarrollaron más tarde el sistema operativo Unix, programándolo exclusivamente en C, incluidas las herramientas necesarias para su funcionamiento, como el compilador. Unix es la influencia más grande conocida en el tema de los sistemas operativos y, unido a él, el lenguaje C es el referente primario en el área de los lenguajes de programación. Más ahora que el sistema operativo GNU/Linux, derivado directo de Unix y también programado en C, ha tenido un repunte impresionante, respaldado por los movimientos de software abierto y libre.

La belleza del lenguaje C radica en que es deliberadamente simple, apegado a los aspectos fundamentales del hardware, esto es, las características del lenguaje mapean directamente a las características de hardware.

Dennis Ritchie describió a C como "*un lenguaje fuertemente tipado, pero débilmente verificado*"; esto es, C es muy estricto en tiempo de compilación, pero muy débil o *demasiado permisivo* en tiempo de ejecución. Lo anterior no es un efecto premeditado del diseño del lenguaje, se debe a las características tan restrictivas que en esos tiempos tenía el hardware.

Finalmente, Bjarne Stroustrup diseñó e implementó el lenguaje de programación C++ en el mismo Centro de Investigación en Ciencias Computacionales de los Laboratorios Bell, en Murray Hill, donde nació el lenguaje C (y a unos cuantos pasos de la oficina de Dennis Ritchie). C++ es un lenguaje de programación de propósitos múltiples, con una cierta orientación hacia la programación de sistemas, que es un *mejor C* y que soporta la abstracción de datos, la POO y la programación genérica. Conceptos que, en los tiempos en que nació C++, se consideraban demasiado

[v] Los sistemas operativos, los compiladores, los intérpretes, etc.

costosos en cuanto a recursos de cómputo, como para usarlos en el mundo real, o demasiado complicados para que los programadores *ordinarios* los aprendieran.

El trabajo de desarrollo de C++ inició en 1979, generando una versión comercial en 1985. Posteriormente, Stroustrup siguió desarrollando el lenguaje con ayuda de sus compañeros de los laboratorios Bell, y de gente externa a estos, hasta que comenzó su estandarización oficial en 1990. A partir de ahí y teniendo como líder del esfuerzo al propio Stroustrup, la definición de C++ la desarrolló el Instituto Nacional Estadounidense de Estándares (*ANSI – American National Standards Institute*) y, desde 1991, la Organización Internacional de Estándares (*ISO – International Organization for Standardization*). Bjarne Stroustrup sigue dirigiendo el grupo internacional de estandarización a cargo de diseñar e implementar las características nuevas del lenguaje. El primer estándar internacional se ratificó en 1998 (denominado ***C++98***) y, el segundo, en 2011 (denominado ***C++11***)[vi].

El desarrollo más significativo del lenguaje C++, después de su primera década de crecimiento, es la *Librería Estándar de Plantillas* (*STL – Standard Template Library*) creada por Alexander Stepanov. La STL es la librería estándar del lenguaje para manejo de contenedores (estructuras de datos) y los algoritmos que las manipulan.[vii]

El "*C con clases*" mostrado en la Figura 1, fue una síntesis inicial de ideas tomadas de los lenguajes C y Simula, que dio paso a su sucesor, C++.

Las fortalezas principales que han hecho de C y C++ lo que hoy son, no son ni la elegancia, ni las características avanzadas; son su flexibilidad, su rendimiento y su estabilidad. En el mundo siempre cambiante de los sistemas de software, estas son tres de las características más codiciadas.

Para finalizar este apartado, cabe señalar que el lector interesado en profundizar en la descripción, el diseño, la evolución, el futuro y la propia programación en este lenguaje puede recurrir a *The C++ Programming Language*,[3] al estándar oficial

[vi] Denominado ***C++0x*** hasta poco antes de la ratificación definitiva en el año 2011.

[vii] La importancia de esta librería radica en que el manejo de los datos es genérico, es decir, es una sola implementación para datos de cualquier tipo, incluyendo tipos definidos por el usuario y, entre estos, los objetos creados a partir de las clases de la POO.

C++11 de la ISO,[4] a *The Design and Evolution of C++*,[5] a la página web del propio Bjarne Stroustrup[6] y a *Programming and Practice – Principles and Practice Using C++*.[1]

1.2. Primer ejemplo: *cout* y *cin*, *printf* y *scanf* al estilo C++

El primer ejemplo de programación que desarrollaremos para introducirnos en el lenguaje C++, será el cálculo del *índice de masa corporal* (*IMC*)[7] para una persona en específico. El IMC es una medida de asociación entre el peso y la talla de un individuo, que se ha utilizado como un medio para evaluar su estado de nutrición.

1.2.1. El ciclo de desarrollo de software

Siguiendo el ciclo de desarrollo de software, comenzaremos con la definición o planteamiento del problema:

> *Problema 1.1.- Escriba un programa que, dado el peso y la altura de una persona, calcule su IMC. El peso deberá ser proporcionado en kilogramos, sin parte fraccional, y la altura en centímetros, también sin parte fraccional. El cálculo del IMC se realiza de acuerdo a la fórmula:* $IMC = \frac{peso}{altura^2}$, *donde el peso sigue estando en kilogramos pero, la altura, debe estar dada en metros. Finalmente, el programa deberá clasificar el IMC calculado y emitir un mensaje acorde al resultado, considerando lo siguiente:*
>
> - *Un* $IMC < 18.5$ *indica que el individuo tiene un peso muy bajo.*
> - *Un* $IMC \geq 18.5$ *y* < 25.0 *indica que el individuo tiene un peso normal.*
> - *Un* $IMC \geq 25.0$ *y* < 30.0 *indica que el individuo sufre de sobrepeso.*
> - *Un* $IMC \geq 30.0$ *indica que el individuo sufre de obesidad.*

El paso siguiente consiste en esquematizar la solución y transformar dicho esquema en un algoritmo que solucione el problema. Para tal fin, emplearemos la estrategia del diseño estructurado de algoritmos denominada *diseño descendente con descomposición (de problemas) por refinamientos sucesivos*.[viii,8] Entonces, el primer nivel (o *cima*) de la estrategia de diseño, en pseudocódigo, consiste en un solo enunciado que expresa la solución completa:

```
Calcular y clasificar el IMC de un individuo, dados su peso y altura.
```

Como se puede observar, realmente esta es una solución completa a nuestro problema; desafortunadamente, con este nivel de abstracción no es posible transformar el esquema anterior en un algoritmo que se pueda implementar en un lenguaje de computadoras. Por tal motivo, comenzamos el proceso de refinamiento dividiendo nuestro primer nivel del diseño en tareas más pequeñas, y las organizamos de acuerdo al orden en que deben ser realizadas para alcanzar la solución. Esto nos lleva al segundo nivel de diseño, el cual se presenta a continuación:

```
Solicitar datos de entrada
Procesar los datos
Presentar resultados
```

En la generalidad de los casos, cuando se emplea la programación estructurada, el segundo nivel de diseño (o el primer refinamiento de la solución) siempre consistirá de estos tres subproblemas referentes a los datos: la entrada, el procesamiento y la salida (o presentación de resultados).

[viii] Esta estrategia busca dividir el problema original en subproblemas más pequeños, de tal manera que sean más fáciles de resolver; posteriormente, cada subproblema se divide nuevamente en problemas más pequeños, y así sucesivamente, hasta obtener problemas *suficientemente* pequeños, que puedan ser expresados en términos de la computadora utilizada o del pseudocódigo o lenguaje de programación empleados. Cabe señalar que cada *nivel de descomposición* en subproblemas, debe ser una solución **completa** al problema originalmente planteado.

Continuando con el proceso de refinamiento, llegamos al tercer nivel de diseño, el cual ya incluye el manejo de variables en concreto. La instrucción en pseudocódigo:

```
Solicitar datos de entrada
```

se puede refinar como:

```
Mostrar "Proporcione su peso (kg) y su altura (cm) sin decimales:"
Leer el peso y la altura del individuo
```

La instrucción en pseudocódigo:

```
Procesar los datos
```

requiere, en primer lugar, la conversión de unidades de la altura del individuo, ya que el dato se recibe en centímetros pero la fórmula del IMC la requiere en metros. Posteriormente, se calcula el IMC con la fórmula dada. Luego, el pseudocódigo queda de la manera siguiente:

```
altura ← altura / 100.0
```

$\texttt{IMC} \leftarrow \frac{peso}{altura^2}$

Finalmente, la instrucción en pseudocódigo:

```
Presentar resultados
```

requiere varias estructuras condicionales anidadas que determinen el rango de clasificación del IMC del individuo en análisis. A diferencia de la forma en que se expresan los rangos en el planteamiento del problema, en el desarrollo del algoritmo optimizamos las instrucciones condicionales comenzando por el rango más grande del IMC, y luego vamos descendiendo. Lo anterior implica que solo probamos un valor límite en cada condición. Nótese también, que una estructura condicional del tipo *EN CASO DE* (o `switch` en los lenguajes de programación C y C++), ***no*** es útil en la presente situación, dado que estamos empleando rangos de valores. Así pues, el refinamiento de la instrucción anterior queda como:

```
SI IMC >= 30.0 ENTONCES
   mensaje ← "Usted sufre de obesidad, tome medidas precautorias."
SI NO
   SI IMC >= 25.0 ENTONCES
         mensaje ← "Usted tiene sobrepeso, cuide su alimentación."
   SI NO
         SI IMC >= 18.5 ENTONCES
               mensaje ← "Felicidades, su peso es normal."
         SI NO // IMC < 18.5
               mensaje ← "Cuidado, su peso es demasiado bajo."
         FIN SI
   FIN SI
FIN SI

Mostrar "Su IMC es: ", IMC, ". ", mensaje
```

Como no existe una convención generalizada para escribir comentarios (anotaciones) en pseudocódigo, nosotros emplearemos dos diagonales ("**//**") para indicar que, a partir de ahí y hasta el final de la línea, el texto que sigue a las diagonales es un comentario, con el único valor de servir de documentación al algoritmo[ix]. En el pseudocódigo anterior, la última estructura condicional:

```
SI NO // IMC < 18.5
```

debiera comprobar si el IMC < 18.5 pero, dadas las comprobaciones anteriores, esta es la única opción posible al llegar a este punto del algoritmo. Por tal motivo, resultaría ineficiente realizar la comprobación explícitamente pero, a fin de recordar dicha situación, colocamos un comentario que lo indique.

El tercer nivel completo del diseño, queda como a continuación se presenta:

```
Mostrar "Proporcione su peso (kg) y su altura (cm) sin decimales: "
Leer el peso y la altura del individuo
```

[ix] Como se verá más adelante, esta es la misma convención que utiliza el lenguaje C++ para definir comentarios de una sola línea, es decir, el texto escrito después de las dos diagonales (y hasta el final de la línea) se considera un comentario.

```
altura ← altura / 100.0
IMC ← peso / altura^2

SI IMC >= 30.0 ENTONCES
   mensaje ← "Usted sufre de obesidad, tome medidas precautorias."
SI NO
   SI IMC >= 25.0 ENTONCES
         mensaje ← "Usted tiene sobrepeso, cuide su alimentación."
   SI NO
         SI IMC >= 18.5 ENTONCES
               mensaje ← "Felicidades, su peso es normal."
         SI NO // IMC < 18.5
               mensaje ← "Cuidado, su peso es demasiado bajo."
         FIN SI
   FIN SI
FIN SI

Mostrar "Su IMC es: ", IMC, ". ", mensaje
```

Llegados a este punto del refinamiento, tenemos ya el detalle suficiente para pasar a la etapa siguiente del ciclo de desarrollo de software, probar que el algoritmo creado sea correcto. Para realizar esta etapa, tomaremos ciertos datos de entrada (pesos y alturas) y calcularemos el resultado *manualmente*. Posteriormente, compararemos los resultados calculados contra los resultados que obtengamos de la *ejecución de escritorio*[x] que hagamos del algoritmo, con los mismos datos de prueba.

kg.	cm.	Cálculo manual	Ejecución de escritorio
50	172	IMC: 16.9010	Su IMC es: 16.9010. Cuidado, su peso es demasiado bajo.
70	172	IMC: 23.6614	Su IMC es: 23.6614. Felicidades, su peso es normal.

[x] Este es un proceso en el que el programador sigue la lógica del algoritmo paso a paso, tal y como la computadora lo haría, verificando que cada instrucción realice lo que se supone debe realizar. Además, se lleva un registro en papel de los valores de las variables más importantes y de todos los cambios que sufren mientras se sigue el algoritmo.

80	172	IMC: 27.0416	Su IMC es: 27.0416. Usted tiene sobrepeso, cuide su alimentación.
90	172	IMC: 30.4218	Su IMC es: 30.4218. Usted sufre de obesidad, tome medidas precautorias.

Nótese que la ejecución de escritorio no se ha detallado completamente, siguiendo cada paso del algoritmo y llevando el control por escrito de las variables principales, esto se deja como ejercicio para que el lector repase sus conocimientos sobre el área. Así también, se puede observar que los resultados del cálculo manual y los de la ejecución de escritorio son idénticos, lo que indica que, al menos para estos valores de prueba, el algoritmo realiza la labor encomendada de manera correcta.

El paso siguiente del ciclo de desarrollo del software consiste en codificar el algoritmo desarrollado en un lenguaje de programación. En nuestro caso, la codificación de nuestra solución al cálculo del IMC en el lenguaje C++, se presenta a continuación:

```
 1.  // Listado 1: imc.cpp
 2.  // Calcula el índice de masa corporal (IMC) de una persona con base
 3.  // en su peso y altura, y emite un mensaje acorde al resultado.
 4.  #include <iostream>
 5.  #include <string>
 6.
 7.  int main(void)
 8.  {
 9.    int peso = 0;
10.
11.    // Leer datos de entrada
12.    std::cout << "Proporcione su peso (kg) sin decimales: ";
13.    std::cin >> peso;
14.
15.    int altura = 0;
16.    std::cout << "Proporcione su altura (cm) sin decimales: ";
17.    std::cin >> altura;
18.
19.    float altura_metros = altura / 100.0;
20.    float IMC = peso / (altura_metros * altura_metros);
21.
22.    std::string mensaje = "";
23.
24.    if (IMC >= 30.0)
25.    {
26.      mensaje = "Usted sufre de obesidad, tome medidas precautorias.";
```

```
27.     }
28.     else
29.     {
30.       if (IMC >= 25.0)
31.       {
32.         mensaje = "Usted tiene sobrepeso, cuide su alimentación.";
33.       }
34.       else
35.       {
36.         if (IMC >= 18.5)
37.         {
38.           mensaje = "Felicidades, su peso es normal.";
39.         }
40.         else // IMC < 18.5
41.         {
42.           mensaje = "Cuidado, su peso es demasiado bajo.";
43.         }
44.       }
45.     }
46.
47.     // Mostrar resultados
48.     std::cout << "\nSu IMC es: " << IMC << ". " << mensaje <<
    std::endl;
49.
50.     return 0;
51.   }
```

Como se mencionó anteriormente, en C++ los comentarios inician con dos diagonales (“**//**”) y finalizan al terminar la línea. También siguen existiendo los comentarios de líneas múltiples estilo C, delimitados por los caracteres “**/***” y “***/**”, aunque la convención general es emplear el nuevo estilo de línea única. Esto lo podemos ver ejemplificado en las líneas 1 a 3 del Listado 1.

En las líneas 4 y 5 del listado anterior, podemos ver la inclusión de dos archivos de cabecera de C++. A diferencia del lenguaje C, en C++ es de uso común no colocar la extensión “**.h**” característica de este tipo de archivos; el compilador la añade automáticamente, si no está presente.

1.2.2. *cin*, *cout* y *cerr*: entrada, salida y error estándar al estilo C++

El empleo de la programación orientada a objetos ha motivado que todo el manejo de la entrada y salida, de teclado y a pantalla, tenga una forma

completamente nueva. Así, el archivo de cabecera `<stdio.h>` ha sido desplazado[xi] por `<iostream>` y, las funciones `scanf/printf`, por los objetos del flujo de entrada estándar (`cin`) y del flujo de salida estándar (`cout`), en conjunción con los operadores de extracción de flujo (>>) y de inserción de flujo (<<). Por el momento consideraremos el término *objeto*, como un sinónimo de *variable*, aunque más adelante estableceremos la diferencia entre ambos. Así también, los descriptores de archivo `stdin` (la entrada estándar), `stdout` (la salida estándar) y `stderr` (la salida estándar de error) han sido vueltos a concebir como flujos (*streams*) de datos, los cuales son, como antes se indica, manejados por medio de objetos/variables.

1.2.3. Una adición al lenguaje: el tipo *string*

El otro archivo de cabecera, `<string>`, contiene la definición de la *clase* de objetos `string` (cadena de caracteres) de la biblioteca estándar de C++. La clase string es una de las novedades en el estándar C++11 y permite el manejo de objetos/variables de este tipo, de una forma tan simple como el manejo de las variables de los tipos de datos básicos, como `int`. Sin embargo, la diferencia entre el tipo string y los tipos básicos es notoria, lo cual quedará de manifiesto cuando entremos al tema de clases y objetos en la POO.

1.2.4. El punto de entrada: la función *main()*

Las líneas 7 a 51 presentan el código fuente de la función `main()`, cuyo *cuerpo* está delimitado por los caracteres "**{**" y "**}**". Esta función, al igual que en C, es el

[xi] La entrada/salida (E/S) estilo C aún existe, pero se recomienda usar la nueva E/S, estilo C++. De la misma manera, los archivos de cabecera referentes al lenguaje C, pueden ser utilizados e incluidos en un programa en C++, con la notación tradicional o, como se aconseja, iniciando con una letra "`c`" y sin colocar la extensión "`.h`". Así pues, por ejemplo, en un programa en C++ el archivo `<stdio.h>` se incluiría como `<cstdio>`.

punto de entrada e inicio de todo programa y, siempre, existirá en todo programa escrito en C++. La línea 7 es la cabecera de la función y nos indica el tipo devuelto por ella, su nombre y, entre paréntesis, los parámetros que recibe. La palabra reservada `void`, entre los paréntesis de parámetros, indica que `main()` no recibe parámetro alguno, al menos en este programa.

1.2.5. Variables, tipos de datos y sufijos de tipo

En las líneas 9, 15, 19, 20 y 22, tenemos la definición e inicialización de variables. Al igual que en C, podemos definir una variable e inicializarla en una misma instrucción pero, a diferencia de este, estas instrucciones se pueden (y se sugiere hacerlo así) colocar en cualquier punto *antes* del primer uso de la variable definida (donde la sintaxis permita una instrucción, claro está), y no solamente después de la apertura de un bloque de código.[xii] Como ya se sabe, la definición de una variable incluye, al menos, el tipo de datos que puede almacenar y su nombre. Los tipos de datos de C++ forman un superconjunto de los tipos de datos del lenguaje C y se muestran en la tabla siguiente junto con el rango de valores que pueden almacenar.[xiii] Cabe señalar que las clases que se pueden definir en la POO son en realidad tipos definidos por el usuario, a partir de los cuales se pueden declarar y definir variables, lo que en teoría permite un infinito número de tipos de datos.

Tipo	Rango de valores	Tamaño
bool	false, true	1 byte

[xii] Un bloque de código es aquel conjunto de una o varias instrucciones que inicia con el caracter "**{**" y termina con el caracter "**}**".

[xiii] Los rangos de valores y tamaños mostrados corresponden a un equipo de cómputo típico con un procesador de 32 bits. Los archivos de cabecera `<climits>` y `<cfloat>` señalan los rangos y tamaños específicos para cada sistema de cómputo, para los tipos enteros y los de punto flotante, respectivamente.

Tipo	Rango de valores	Tamaño
char	0 … 255	1 byte
signed char	-128 … +127	1 byte
unsigned char	0 … 255	1 byte
short int	-32,768 … +32,767	2 bytes
unsigned short int	0 … 65,535	2 bytes
int[xiv]	–2,147,483,648 … +2,147,483,647	4 bytes
unsigned int	0 … 4,294,967,295	4 bytes
long int	–2,147,483,648 … +2,147,483,647	4 bytes
unsigned long int	0 … 4,294,967,295	4 bytes
long long int	–9,223,372,036,854,775,808 … +9,223,372,036,854,775,807	8 bytes
unsigned long long int	0 … +18,446,744,073,709,551,615	8 bytes
float	1.17549×10^{-38} … $3.40282\times10^{+38}$, 6 dígitos de precisión	4 bytes
double	2.22507×10^{-308} … $1.79769\times10^{+308}$, 15 dígitos de precisión	8 bytes
long double	2.22507×10^{-308} … $1.79769\times10^{+308}$, 15 dígitos de precisión	8 bytes

El tipo `bool` enlistado en la tabla anterior, es una adición proporcionada por C++, la cual permite que una variable almacene exclusivamente los valores **`true`** (verdadero) y **`false`** (falso).[xv] Sin embargo, con la finalidad de mantener la compatibilidad hacia atrás con el lenguaje C, se mantiene la convención de este lenguaje en la que el valor **`0`** representa el valor lógico **falso** y, cualquier otro valor diferente de 0, representa el valor lógico **verdadero**.

C++ proporciona varios caracteres o sufijos que sirven para indicar el tipo de una constante que inicializa una variable, es decir, el tipo de un número que se escribe literalmente, en la inicialización de una variable. Los sufijos enteros son: **`u`** o **`U`** para un entero sin signo, **`l`** o **`L`** para un entero largo, **`ul`** o **`UL`** para un entero largo sin

[xiv] Con excepción del tipo de datos `int`, los tipos integrales pueden omitir en su nombre la palabra reservada `int`. Por ejemplo, el tipo puede escribirse como `short int` o solamente como `short`.

[xv] **`true`** y **`false`** son palabras reservadas del lenguaje C++.

signo, **ll** o **LL** para un entero largo largo y **ull** o **ULL** para un entero largo largo sin signo.

Si una constante entera no tiene sufijo, su tipo será entero; si la constante no puede almacenarse en un entero, será de tipo entero largo; y, si tampoco puede almacenarse en un entero largo, será de tipo entero largo largo.

Los sufijos para los tipos de punto flotante son: **f** o **F** para un número de tipo `float` y **l** o **L** para un número de tipo `long double`. Una constante de punto flotante que no tenga sufijo será de tipo `double`.

Los ejemplos siguientes emplean sufijos para indicar el tipo de datos pretendido:

```
unsigned int orden = 18u;
long alumnos = 55000L; // "L" evita confusión entre 1 (uno) y l (ele)
unsigned long long habitantes = 7000000000ULL;
float piRedondeada = 3.1416f;
long double pi = 3.141592653589793L;
```

1.2.6. Entrada y salida basada en flujos (*streams*) y espacios de nombres (*namespaces*)

La línea 12:

```
std::cout << "Proporcione su peso (kg) sin decimales: ";
```

es una instrucción de *salida*, equivalente a una llamada a la función `printf` del lenguaje C. En esta instrucción le indicamos al programa que envíe un mensaje de texto, mediante el operador de inserción de flujo "**<<**", al objeto del flujo de la salida estándar `cout`. Por defecto, dicho flujo está conectado a la pantalla, de tal manera que el resultado real es que podemos ver el mensaje impreso en ella.

La notación "`std::cout`" emplea el operador de resolución de ámbito "**::**", el cual nos señala que el nombre del identificador/variable a su derecha, en este caso "`cout`", pertenece al *espacio de nombres* a su izquierda, en este caso "`std`".

Empleando una analogía lo anterior quiere decir que, tanto el nombre cout como todos los nombres que sean precedidos por "`std::`", pertenecen a una misma *familia*, cuyo *apellido* es "`std`". Es decir, cuando un equipo de trabajo se encuentra desarrollando un programa, debe definir todos los identificadores que utilizará, pero sin duplicar ni uno solo de ellos. Así, si sucede que el programador encargado de desarrollar el código de manejo de archivos utiliza una función "`busca()`" para encontrar un dato en un archivo y, el programador encargado de desarrollar el código de la interfaz gráfica de usuario, también emplea una función "`busca()`", pero para localizar un botón donde el usuario haya dado clic con el ratón; se generará un conflicto de identificadores cuando el equipo de desarrollo una ambas partes del código. El programa no podrá determinar cuál de las dos funciones se debe ejecutar cada vez que encuentre una llamada a la función "`busca()`".

Para evitar el problema anterior, C++ crea los espacios de nombre, los cuales proporcionan un *apellido* a cada identificador. Así pues, en el ejemplo anterior, el primer programador podría definir su propio espacio de nombres "*archivo*" y, el segundo, "*interfaz*"; de tal manera que la función "`busca()`" del primer programador tendría el nombre "`archivo::busca()`" y, la del segundo programador, el nombre "`interfaz::busca()`", solucionando el conflicto de nombres.

Entonces, para evitar conflictos entre el propio C++ y los programadores que lo emplean, el lenguaje les proporciona el *apellido* "`std`" (el espacio de nombres estándar –*std*) a todos los identificadores que usa, diferenciándolos de los utilizados por cualquier programador.

En el Listado 1 podemos ver que los nombres pertenecientes a C++ son: `cout`, `cin`, `string` y `endl`.

En contraparte a la instrucción de salida en la línea 12, la línea 13:

```
std::cin >> peso;
```

es una instrucción de *entrada*, equivalente a una llamada a la función `scanf` del lenguaje C. En esta instrucción le indicamos al programa que reciba o lea un flujo de caracteres, mediante el operador de extracción de flujo "**>>**", del objeto del flujo de entrada estándar `cin`. Por defecto, dicho flujo está conectado al teclado, de tal manera que el resultado real es que leemos el valor que queremos asignar a una variable.

La fortaleza de la E/S basada en flujos de C++ proviene del hecho de que está sobrecargada, es decir, que funciona para cualquiera de los tipos de datos básicos. Por ejemplo, en las líneas 13 y 17 leemos el peso y la altura de una persona como datos de tipo entero, pero sin indicarlo de manera explícita en momento alguno. C++ sabe qué tipo de dato leer, al *ver* el tipo de la variable cuyo valor solicitamos desde el teclado. Lo mismo funciona para la impresión en pantalla de una variable, no es necesario indicar el tipo de la misma ya que C++ determinará el tipo a imprimir, a partir del tipo de la variable cuyo valor deseamos presentar en pantalla, por ejemplo el IMC de tipo `float` en la línea 48 del Listado 1.

1.2.7. Conversión entre tipos de datos

Continuando con el análisis, en la línea 19 se declara e inicializa la variable `altura_metros` mediante la conversión del valor proporcionado por el usuario, en centímetros, a metros. En esta instrucción es muy importante recordar que si el valor `100.0` se hubiera escrito solamente como `100`, de acuerdo con el apartado anterior del uso de sufijos para indicar el tipo de una variable, C++ lo habría interpretado como un valor entero y, considerando además que la variable `altura` es de tipo entero, se habría realizado una *división entera*, truncando la parte fraccional del valor resultante aún y cuando la variable que recibe el resultado, `altura_metros`, sea de tipo `float`.

En este sentido, al escribir `100.0` C++ interpreta dicho valor como de tipo `double`, por lo que promueve a este tipo de dato también la variable `altura` y,

solamente después, realiza la división de punto flotante.[xvi] Finalmente, se convierte implícitamente el resultado de la división a un valor del mismo tipo de la variable `altura_metros`, en este caso `float`, que es a quien se asigna el resultado.

1.2.8. Expresiones y operadores

Las líneas 19 y 20 del Listado 1 son instrucciones de cálculo aritmético o *expresiones aritméticas*. C++, al igual que C, permite realizar operaciones de este tipo pero, además, de comparación, de asignación, lógicas, de bits, de incremento o decremento, sobre apuntadores, etc.

Otra cuestión importante a considerar con el uso de expresiones es que, a diferencia de las expresiones matemáticas, en C y C++ las expresiones se deben escribir en línea horizontal (cuya escritura se puede dividir en varios renglones). Esto, generalmente, se logra añadiendo paréntesis adicionales para agrupar subexpresiones. Para ejemplificar lo anterior, a continuación se presenta la forma matemática de escritura de varias expresiones así como la notación de línea horizontal correspondiente:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

```
x1 = (-b + sqrt((b * b) - (4 * a * c))) / (2 * a);
x2 = (-b - sqrt((b * b) - (4 * a * c))) / (2 * a);
```

$$A = \pi r^2$$

```
A = 3.1416 * r * r;
```

[xvi] Recuérdese que el tipo de cada valor (constante o variable) en una expresión con *mezcla de tipos* (expresiones que contienen valores de más de un tipo de datos) se promueve al tipo de datos *más grande* (en realidad, se crean versiones temporales de los valores, los datos originales no se alteran). Esto es, si la expresión contiene un dato de tipo `long double`, todos los demás valores se promoverán a este tipo; si contiene un dato de tipo `double`, todos los demás se convertirán a este tipo; esto sigue aplicándose siguiendo la secuencia `float`, `unsigned long long int`, `long long int`, `unsigned long int`, `long int`, `unsigned int`, `int`, `unsigned short int`, `short int`, `unsigned char`, `char` y, finalmente, el tipo de datos `bool`.

$$r = \frac{1}{2}at^2 + v_o t + r_0$$

```
r = ((1/2) * a * t * t) + (v0 * t) + (r0);
```

$$s = \frac{a}{b} \times \frac{c}{d}$$

```
s = (a/b) * (c/d);
```

Toda expresión está conformada por variables, constantes y/u operadores. La tabla siguiente presenta, en orden decreciente de *prioridad*, los operadores que proporciona el lenguaje de programación C++. Así también, se indica su tipo y la *asociatividad* que rige a cada uno de ellos.

Operador	Tipo	Asociatividad
`::`	Resolución de ámbito binaria	Izquierda a derecha
`()`	Paréntesis de agrupación	
`()`	Llamada a función	Izquierda a derecha
`[]`	Subíndice de arreglo	
`.`	Selección de un miembro mediante un objeto	
`->`	Selección de un miembro mediante un apuntador	
`++, --`	Incremento y decremento unario posfijo	
`static_cast<>()`	Coerción de tipo verificado en tiempo de compilación	
`++, --`	Incremento y decremento unario prefijo	Derecha a izquierda
`+, -`	Más y menos unario	
`!`	Negación lógica unaria	
`~`	Complemento unario de bits	
`sizeof`	Determinar tamaño en bytes	
`&`	Dirección de una variable	
`*`	Desreferencia	
`(tipo)`	Coerción de tipo estilo C	
`.*`	Apuntador a un miembro mediante un objeto	Izquierda a derecha
`->*`	Apuntador a un miembro mediante un apuntador	
`*, /, %`	Multiplicación, división y módulo	
`+, -`	Suma y resta	
`<<, >>`	Corrimiento de bits a la izquierda y a la derecha	

Operador	Tipo	Asociatividad
`<, <=, >, >=, ==, !=`	Menor que, menor o igual que, mayor que, mayor o igual que, igual que y no es igual que	
`&, ^, |`	Y (and), o exclusivo (xor) y o inclusivo (or) de bits	
`&&, ||`	Y (and) y o (or) lógicos	
`?:`	Condicional ternario	Derecha a izquierda
`=`	Asignación	
`+=, -=, *=, /=, %=`	Asignación de suma, resta, multiplicación, división y módulo	
`&=, ^=, |=`	Asignación de y (and), o exclusivo (xor) y o inclusivo (or) de bits	
`<<=, >>=`	Asignación de corrimiento de bits a la izquierda y a la derecha	
`,`	Coma	Izquierda a derecha

La prioridad indica el orden en que se deben realizar las operaciones dentro de una expresión. Por ejemplo, 3 + 2 * 4 resulta 11, ya que la multiplicación tiene mayor prioridad que la suma. Esto es equivalente a escribir 3 + (2 * 4).

La asociatividad indica el orden en que se deben realizar las operaciones indicadas por varios operadores que tienen la misma prioridad. Por ejemplo, la expresión 10 / 5 * 2 tiene como resultado 4, ya que la división y la multiplicación tienen la misma prioridad, pero tienen asociatividad de izquierda a derecha, es decir, se ejecuta primero la operación más a la izquierda, luego la segunda más a la izquierda y así, hasta realizar la operación más a la derecha. Esto es equivalente a escribir (10 / 5) * 2.

Prácticamente todos los operadores en la tabla anterior existen también en el lenguaje C, exceptuando los que mencionan acceso a miembros, que se verán una vez entrados en los temas de la POO. El otro operador en la tabla que proporciona C++ pero no C, es el operador unario "`static_cast<>()`". Para ejemplificar el uso de este operador, retomemos nuevamente la línea 19 del Listado 1:

```
float altura_metros = altura / 100.0;
```

Ya se comentó en párrafos anteriores que si la constante `100.0` se reescribiera como el entero `100`, esto provocaría una división entera y el truncamiento de la parte fraccionaria del resultado. Para evitar este problema y señalar de forma explícita la coerción del tipo entero al tipo flotante, podríamos reescribir la línea anterior como:

```
float altura_metros = static_cast<float>(altura) / 100;
```

El uso del operador `static_cast<`*`tipo`*`>(`*`dato`*`)` crea una copia temporal de su operando entre paréntesis, convertido al tipo indicado entre los paréntesis angulares "**<**" y "**>**". En el ejemplo anterior se crea una copia temporal de la variable `altura`, pero de tipo `float`; posteriormente y siguiendo las reglas de promoción en expresiones con mezcla de tipos, C++ promueve la constante entera `100` al tipo `float` y, finalmente, realiza la división de punto flotante.

Cabe señalar que los lenguajes C y C++ cuentan con el operador de coerción de tipo "**(***`tipo`***)** *`dato`*", pero dado su parecido con la notación de llamadas a función, entre otros motivos, en C++ se prefiere la utilización del operador "`static_cast<>()`", para señalar la coerción de tipos de manera más explícita.

1.2.9. Más sobre el flujo de salida estándar

Para continuar y finalizar el análisis del Listado 1 tenemos que, en la línea 22, se inicializa la variable `mensaje` de tipo `string` con la cadena vacía (**"**) y, en la estructura condicional `if` de las líneas 24-45, se le asigna a esta variable el texto resultante de la clasificación del IMC. Para culminar, los resultados obtenidos se muestran en pantalla (se envían al flujo de salida, representado por el objeto `cout`, mediante el operador de inserción "**<<**") en la línea 48.

```
std::cout << "\nSu IMC es: " << IMC << ". " << mensaje << std::endl;
```

La instrucción anterior muestra la posibilidad que proporciona C++ para concatenar el envío al flujo de salida de varios valores a la vez. En este caso se envía una cadena de texto (“\nSu IMC es:”), una variable de tipo float (IMC), otra cadena de texto (“. ”) y un objeto/variable de tipo string (mensaje). Lo anterior demuestra lo ya mencionado: la fortaleza de la E/S basada en flujos; es decir, que está sobrecargada para trabajar con todos los tipos de datos fundamentales (y, como lo demuestra la instrucción anterior, también para el tipo string).

Al igual que en el lenguaje de programación C, el símbolo ‘****’ es un carácter de *escape* que, en combinación con el carácter que le sigue en una cadena de texto, forman una *secuencia de escape*. Una secuencia de escape indica un carácter que no se puede imprimir en pantalla por tener o representar un significado especial. Por ejemplo, la secuencia de escape “\n” indica un carácter de nueva línea, equivalente a presionar la tecla *Enter/Intro*. La tabla siguiente presenta algunas secuencias de escape comunes.

Secuencia de escape	Descripción
\n	Nueva línea. Posiciona el cursor al inicio de la línea siguiente.
\t	Tabulador horizontal. Mueve el cursor a la parada siguiente del tabulador.
\r	Retorno de carro. Posiciona el cursor al inicio de la línea, sin avanzar a la siguiente.
\a	Alerta. Hace sonar la campana del sistema.
\\	Diagonal invertida. Imprime literalmente un carácter de diagonal invertida.
\’	Apóstrofe. Imprime literalmente un carácter de apóstrofe.
\”	Comillas. Imprime literalmente un carácter de comillas.

En la línea 48, lo último que se envía al flujo de salida, std::endl, se denomina un *manipulador de flujo*. endl significa “*end line*” (“*final de línea*”) y pertenece a la *familia* o espacio de nombres std. La función de este manipulador consiste en enviar al flujo de salida un carácter de nueva línea (“\n”) y vaciar el buffer asociado a dicho flujo. Es decir que, en los sistemas en los cuales se almacena la salida que será impresa en pantalla hasta el momento en que se acumula

suficiente como para ameritar que se imprima, `std::endl` fuerza el vaciado inmediato del buffer y, por lo tanto, fuerza la impresión inmediata en pantalla del texto acumulado hasta ese momento.[xvii]

Por último, la instrucción en la línea 50:

```
return 0;
```

indica que el valor devuelto por la función `main()` a quien la llama (el sistema operativo) será `0`, un valor de tipo `int` como la cabecera de la función en la línea 7 lo señala. Este valor es la convención generalizada para indicar que el programa terminó de manera satisfactoria y sin problemas.

1.3. Estructuras de control

En 1966,[9] Corrado Böhm y Giuseppe Jacopini demostraron que toda función computable[xviii] puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) específicamente son:

- La **secuencia**: ejecución de una instrucción tras otra.
- La **selección**: ejecución de una de dos instrucciones (o conjuntos de instrucciones), según el valor de una variable booleana o condición.

[xvii] Esto puede ser importante, por ejemplo, cuando queremos imprimir en pantalla un mensaje solicitando al usuario que introduzca uno o varios valores.

[xviii] Permítasenos igualar este término al de *programa*, aunque esto no sea totalmente preciso.

- **Iteración**: ejecución de una instrucción o de un conjunto de instrucciones mientras una variable booleana, o condición, sea *verdadera*. Esta

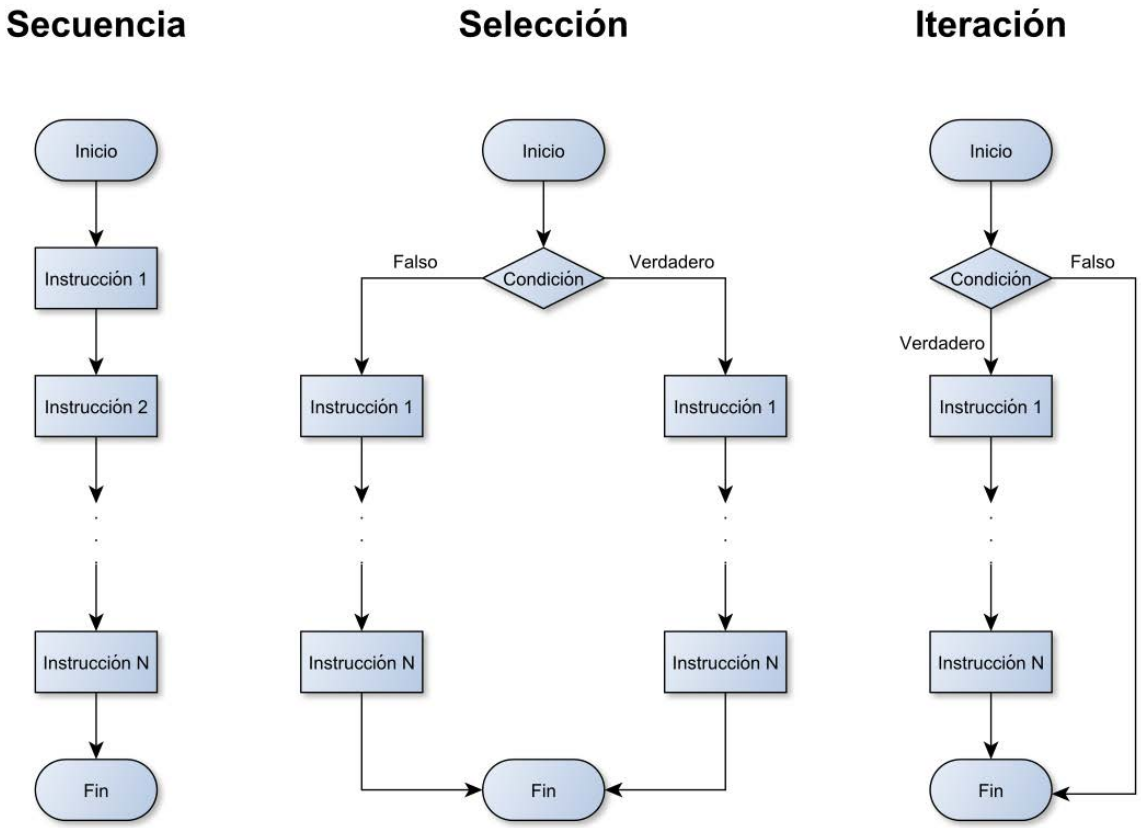

Figura 2. Diagramas de flujo de las estructuras de control

estructura lógica también se conoce como ciclo.

La secuencia es una estructura de control que existe implícitamente en cualquier lenguaje de programación estructurada, o en la parte estructurada de un lenguaje *híbrido* como C++ (híbrido por soportar la programación estructurada y la POO). Esto es, como la Figura 2 lo representa, la secuencia implica ejecutar una instrucción después de otra, tal y como lo realiza C++ de forma predeterminada.

La selección es implementada en C++ por medio de:

- Una instrucción `if` de elección simple (la condición es verdadera).

- Una instrucción `if…else` de elección doble (la condición es falsa o verdadera).
- La instrucción `switch` de selección múltiple.

La iteración es implementada en C++ por medio de:

- La instrucción `while`.
- La instrucción `do…while`.
- La instrucción `for`.

A partir de aquí, cualquier programa estructurado en C++[xix] se construye siguiendo las reglas siguientes:

1. Iniciar con un diagrama de flujo que contenga un símbolo terminal de inicio, un símbolo de proceso y un símbolo terminal de fin. Lo cual denominamos, para simple referencia, como **regla inicial**.
2. Todo símbolo de proceso se puede reemplazar por dos símbolos de proceso en secuencia. **Regla de apilamiento**.
3. Todo símbolo de proceso se puede reemplazar por cualquier instrucción de control (`if`, `if…else`, `switch`, `while`, `do…while` o `for`). **Regla de anidamiento**.
4. Las reglas de apilamiento y anidamiento se pueden aplicar en cualquier orden y tantas veces como se requiera. **Regla de generación**.

Por ejemplo, la evolución del desarrollo del programa estructurado del Listado 1 se muestra en las Figuras 3 y 4.

[xix] La sintaxis de las instrucciones `if`, `if…else`, `switch`, `while`, `do…while` y `for`, es idéntica en los lenguajes de programación C y C++, razón por la cual no se tratan a detalle en esta obra, pero se muestran ejemplos de su utilización en cada uno de los programas desarrollados.

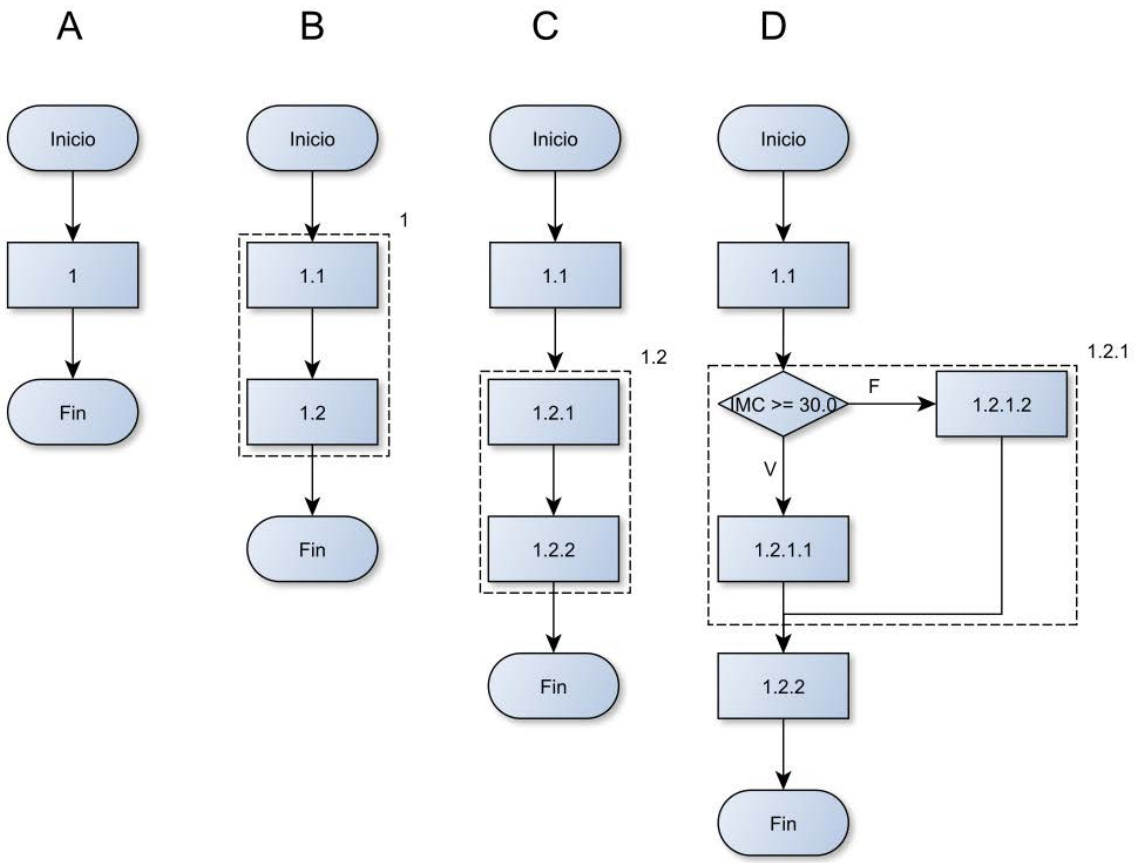

Figura 3. Evolución del desarrollo del programa estructurado del Listado 1. Parte 1 de 2

En la Figura 3-A comenzamos con la regla inicial. En la Figura 3-B aplicamos la regla de apilamiento al símbolo de proceso 1 de la Figura 3-A, generando los símbolos de proceso 1.1 y 1.2. En esta figura, se indica el símbolo de proceso 1 como un recuadro con línea discontinua, marcado en su parte superior derecha con el número correspondiente.

En la Figura 3-C aplicamos nuevamente la regla de apilamiento, ahora al símbolo de proceso 1.2 (indicado en esta figura con un recuadro con línea discontinua) de la Figura 3-B, generando los símbolos de proceso 1.2.1 y 1.2.2.

En la Figura 3-D aplicamos la regla de anidamiento al símbolo de proceso 1.2.1 de la Figura 3-C, generando los símbolos de proceso 1.2.1.1, 1.2.1.2 y el símbolo de decisión "`IMC >= 30.0`".

En la Figura 4 aplicamos nuevamente la regla de anidamiento, ahora al símbolo de proceso1.2.1.2, generando los símbolos de proceso 1.2.1.2.1, 1.2.1.2.2 y el símbolo de decisión "`IMC >= 25.0`".

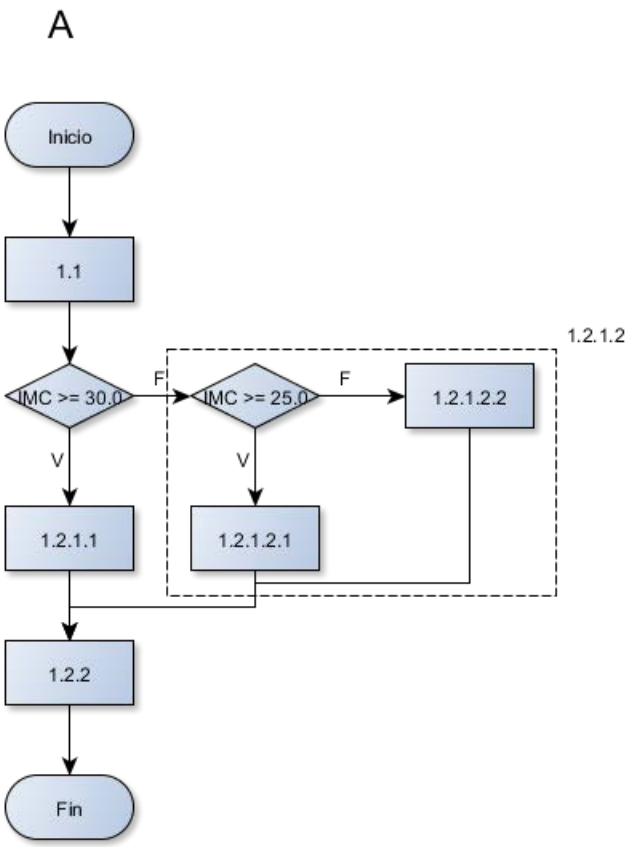

Figura 4. Evolución del desarrollo del programa estructurado del Listado 1. Parte 2 de 2

En este momento ya podemos observar que, a partir de la Figura 4, sí es posible generar el programa estructurado del Listado 1. Aplicando sucesivamente la regla de apilamiento, el símbolo de proceso 1.1 se convertirá en las líneas 9 a 23 del algoritmo en la función `main()`. El símbolo de proceso 1.2.2, también aplicando sucesivamente la regla de apilamiento, se transformará en las líneas 46 a 50. El símbolo de procesamiento 1.2.1 en la Figura 3-C, generará la estructura condicional `if` completa de las líneas 24 a 45. En donde, siguiendo en orden las líneas de código de esta estructura en el Listado 1, la línea 24 la genera el condicional de la Figura 4 "`IMC >= 30.0`"; las líneas 25 a 27 las genera el símbolo de

proceso 1.2.1.1; las líneas 28 a 30 las genera la ruta seguida al condicional de la Figura 4 "`IMC >= 25.0`"; las líneas 31 a 33 las genera el símbolo de proceso 1.2.1.2.1; y las líneas 34 a 45 las generará el símbolo de proceso 1.2.1.2.2, aplicando dos veces la regla de anidamiento.

Los diagramas de flujo construidos aplicando las cuatro reglas anteriores constituyen el conjunto de todos los posibles diagramas de flujo y, por lo tanto, el conjunto de todos los programas estructurados posibles. Estos programas, aparte de ser desarrollados siguiendo el ciclo de desarrollo de software, deben ser implementados en el código fuente de un lenguaje de programación y, posteriormente, *compilados a lenguaje máquina*. Este proceso y su terminología se explican en el apartado siguiente.

1.4. Proceso de compilación y la estrategia de mejor error al compilar que al ejecutar o, en otras palabras, C consiente y C++ fastidia

La *compilación* es el proceso mediante el cual un programa especial, denominado *compilador*, traduce el código *fuente* escrito por un programador en un lenguaje, como C y C++, al único lenguaje que una computadora entiende, el *lenguaje máquina*.[xx]

Estrictamente hablando, la compilación se refiere exclusivamente al proceso de traducción del código fuente al código máquina, pero se ha generalizado comúnmente aceptar que dicho proceso está conformado por las fases que se muestran en la Figura 5 y que se detallan a continuación.

La primera fase consiste en la escritura del código fuente de un programa mediante un editor de texto. El archivo generado tendrá la extensión "`.cpp`", para el código fuente escrito en C++ (o "`.h`" para los archivos de cabecera).

[xx] El lenguaje máquina también es conocido como código objeto, código binario o código ejecutable.

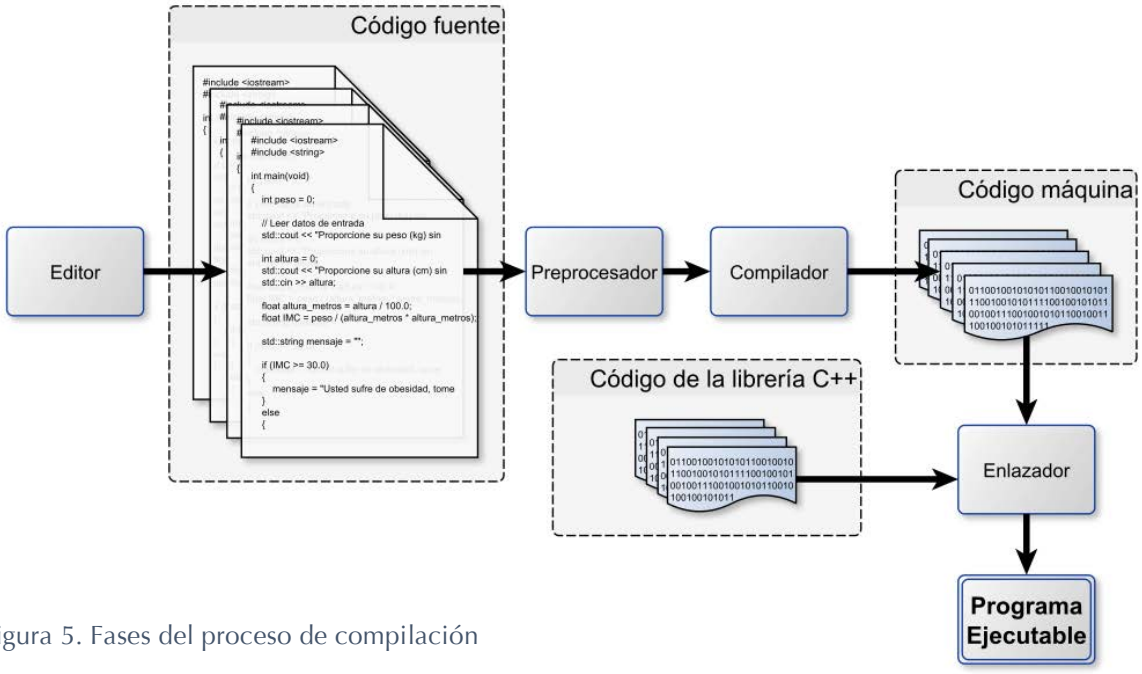

Figura 5. Fases del proceso de compilación

La segunda fase consiste en el preprocesamiento del archivo "`.cpp`", esto es, el módulo de preproceso del compilador tomará el código fuente, buscará todas las líneas que inician con el carácter "**#**" y, de acuerdo a la directiva que corresponda a cada línea encontrada, realizará un proceso específico. Por ejemplo, para la directiva "`#include <iostream>`", el proceso a realizar por parte del preprocesador consiste en buscar el archivo "`iostream.h`" en el sistema de archivos de la computadora, y reemplazar la línea de código fuente de la directiva `include`, por el contenido completo del archivo. Esto se realiza, por supuesto, en una copia del archivo fuente original, el cual no se modifica, y que es la que se pasa a la fase siguiente.

La tercera fase del proceso es, en sí, la traducción a código máquina. En esta fase se genera un archivo con el mismo nombre del archivo "`.cpp`", pero cambiando la extensión a "`.o`" o "`.obj`", dependiendo del compilador utilizado. La extensión se debe a que el archivo generado contiene código objeto o binario.

Aún y cuando la tercera fase de la compilación genera código binario, esto es, código que ya es ejecutable, el uso que los programas realizan de las funciones de la

librería estándar proporcionada por C++ requiere un proceso de *enlazamiento*. Este paso, la cuarta fase, consiste en que el módulo *enlazador* (*linker*) del compilador, toma el código objeto generado por la tercera fase y verifica todas las llamadas que este realiza a funciones de la librería estándar. Sabiendo cuáles son las funciones requeridas, se busca su código objeto en el sistema de archivos de la computadora y se *enlaza* con el código objeto del programador. En otras palabras, el código objeto de la librería se introduce en el código objeto del programador, tal y como si éste hubiera escrito el código fuente de las funciones de la librería estándar en su propio código fuente.[xxi]

Finalmente, el enlazador genera el archivo o programa ejecutable, generalmente con el mismo nombre del archivo "`.cpp`", pero con extensión "`.exe`" o sin ella, dependiendo del sistema operativo donde se compile el programa.

El proceso de desarrollo de software es muy repetitivo, las fases de la compilación se realizarán todas y cada una de las veces que se modifique el código fuente del programa. Por esta razón, es muy conveniente el empleo de un ambiente de desarrollo integrado (*IDE – Integrated Development Environment*), el cual permite llevar a cabo todas las fases del proceso en un solo programa, desde la edición del código fuente, a la propia ejecución del programa ya compilado. No hay magia en el proceso, simplemente es que el IDE utiliza las mismas herramientas referidas, pero libera al programador de este procedimiento engorroso y propenso a errores humanos.

Con relación al proceso de compilación, cabe señalar que todo programador que llega al lenguaje de programación C++, desde el mundo del lenguaje C, encuentra que el compilador de C++ es muy *fastidioso*. Genera una cantidad enorme de errores y advertencias sobre código sospechoso o, aparentemente, mal escrito.

[xxi] Al momento de realizar la instalación de nuestro compilador preferido en nuestra computadora, se configuran todas las rutas necesarias tanto para los archivos de cabecera, como para los archivos de código objeto de la librería estándar. Por este motivo, el compilador siempre *sabe* dónde localizar dichos archivos.

Considerando que el compilador del lenguaje C es muy permisivo, dejando al programador la responsabilidad de saber *perfectamente* lo que está haciendo con cada instrucción de *su* programa, resulta comprensible que al comenzar a desarrollar programas en C++, su compilador *fastidie*. Esta intromisión del compilador, aparentemente innecesaria, tiene una razón de ser muy poderosa, C++ se desarrolló (y se sigue desarrollando) con la meta de enfrentar los proyectos de software cada vez más complejos que el mundo de hoy demanda. Por tal motivo, toda ayuda que este le pueda proporcionar al programador, con la finalidad de reducir la complejidad de los proyectos, debe ser bienvenida muy gratamente. En términos coloquiales: "*Mejor error al compilar, que al ejecutar*". Esto refiere a la gran ayuda que resulta un pequeño mensaje de advertencia emitido por el compilador en cuanto a una instrucción en específico, evitando horas inacabables de depuración en la búsqueda de un error intermitente que aparece solo bajo circunstancias aparentemente aleatorias.

Referencias

[1] Stroustrup, Bjarne. (**2009**) Programming – Principles and Practice Using C++. Kendallville, Indiana: Addison-Wesley.

[2] Rosenberg, Doug and Scott, Kendall. (**2001**) Driving design: the problem domain. Dr. Dobb's. Consultado: 04 de octubre de 2013, de http://www.drdobbs.com/driving-design-the-problem-domain/184414689

[3] Stroustrup, Bjarne. (**2013**) The C++ Programming Language, 4th. Edition. Ann Arbor, Michigan: Addison-Wesley.

[4] ISO/IEC 14882:2003. (**2003**) Programming Languages – C++. The C++ standard.

[5] Stroustrup, Bjarne. (**1994**) The Design and Evolution of C++. Addison-Wesley.

[6] Stroustrup, Bjarne. Welcome to Bjarne Stroustrup's homepage! Consulta 09 de octubre de 2013, de http://www.stroustrup.com/

[7] Wikipedia. "Índice de masa corporal." Consultado: 05 de noviembre de 2013. http://es.wikipedia.org/wiki/%C3%8Dndice_de_masa_corporal

[8] Wirth, N. (**1971**) Program development by stepwise refinement. Communications of the ACM, Volume 14, Issue 4, April 1971, Pages 221-227. Consulta 07 de noviembre de 2013, de http://doi.acm.org/10.1145/362575.362577

[9] Böhm, C. & Jacopini, G. (**1966**) Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Communications of the ACM, Volume 9, Issue 5, May 1966, Pages 366-371. Consulta 15 de noviembre de 2013, de http://dx.doi.org/10.1145/355592.365646

Capítulo 2. Funciones

"… toma unos diez años el desarrollar experiencia en cualquiera de una amplia variedad de áreas, incluyendo jugar ajedrez, la composición musical, el manejo del telégrafo, la pintura, tocar el piano, la natación, el tenis y la investigación en neuropsicología y topología. La clave es la *práctica deliberativa*: no sólo hacerlo una y otra vez, sino retarse a uno mismo con una tarea que esté más allá de nuestra capacidad actual, intentarlo, analizar nuestro rendimiento mientras lo hacemos y después de hacerlo, y corregir cualquier error. Luego hay que repetirlo. Y repetirlo de nuevo…"

Teach Yourself Programming in Ten Years
(Enséñese a usted mismo a programar en diez años)
Peter Norvig

2.1 Introducción

Divide et vinces (*divide y vencerás*) es una frase atribuida al emperador romano Julio César, adoptada (y *popularizada*) por Nicolás Maquiavelo en la época renacentista. Esta frase, atribuida a personajes que no podrían estar más lejos del mundo moderno de la programación, es uno de los pilares que sostienen prácticamente cualquier paradigma de programación.

Con relación a la programación estructurada se aplica en dos vertientes separadas, la primera, en cuanto al diseño descendente con descomposición por refinamientos sucesivos, que empleamos en el capítulo anterior; y, la segunda, en cuanto a la creación de un programa mediante módulos bien diferenciados denominados *funciones*.

Las funciones son definidas como unidades autocontenidas que ayudan a modularizar un programa,[1] también son vistas como *cajas negras* a las que se les proporciona una cierta cantidad de información o datos y que, con base en ellos, calculan un valor que devolverán cuando terminen su proceso.[2] A la información que reciben se le denomina *parámetros* y, al valor que devuelven, *valor de devolución o de retorno*.

El término *función* es empleado tanto en el lenguaje de programación C como en C++, pero otros lenguajes y paradigmas los designan como procedimientos, subprogramas o métodos (este es el término empleado en la POO).[3]

En C y C++ existen dos formas de catalogar a las funciones, aquellas proporcionadas por el lenguaje, conocidas como funciones de la biblioteca estándar, y aquéllas creadas por el programador. Las funciones de la biblioteca estándar requieren la inclusión de un archivo de cabecera, como `iostream`, además de que el proceso

de compilación enlazará el código objeto de dichas funciones, proporcionado por la instalación del compilador que estemos usando en específico, con el código objeto de las funciones desarrolladas por el programador, a fin de crear el programa ejecutable final.

2.2 Medidas de tendencia central: media, mediana y moda[4]

Para continuar con la revisión del paradigma estructurado, ahora en lo concerniente a las funciones, desarrollaremos otro ejemplo de programación. En este ejemplo calcularemos tres de las medidas de tendencia central de un conjunto de datos proporcionado por el usuario.

2.2.1 El ciclo de desarrollo de software

Para desarrollar el software que se nos plantea, continuaremos empleando buenas prácticas de ingeniería de software, por lo que comenzaremos con la primera fase del ciclo de desarrollo, el planteamiento del problema:

> *Problema 2.1.- Escriba un programa que solicite al usuario la introducción de datos numéricos enteros en el rango* $0 \leq x \leq 10$. *La introducción de datos deberá terminar cuando el usuario introduzca un número negativo, pero considerando que el número de datos a introducir será como máximo 25 (bajo este contrato, si el usuario introduce una mayor cantidad, será su responsabilidad que el programa opere de manera incorrecta). Terminada la introducción de datos, el programa deberá calcular e imprimir en pantalla los datos introducidos (ordenándolos de menor a mayor) y las medidas de tendencia central siguientes:*
>
> - ***Media**, **media aritmética** o **promedio**: es el valor obtenido de la suma de un conjunto de valores dividida entre el número de ellos.*

Formalmente: Dado un conjunto numérico de datos, $x_1 + x_2 + x_3 + \cdots + x_n$*, su media aritmética se define como:*

$$\bar{x} = \frac{x_1 + x_2 + x_3 + \cdots + x_n}{n} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- ***Mediana****: es un valor de entre un conjunto de valores que deja por debajo de sí a la mitad de los datos, una vez que estos están ordenados de menor a mayor. Por ejemplo, la mediana de los datos:*

$$3, 4, 2, 3, 2, 1, 1, 2, 1, 1, 2, 1, 1$$

es 2, puesto que, una vez ordenados los datos:

$$1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4$$

el que ocupa la posición central es 2.

$$\underbrace{1, 1, 1, 1, 1, 1}_{Valores\ inferiores}\quad \underbrace{2}_{Mediana}\quad \underbrace{2, 2, 2, 3, 3, 4}_{Valores\ superiores}$$

Si el número de datos fuera impar, la mediana es el promedio de los dos datos intermedios.

- ***Moda****: es el dato más repetido de un conjunto de valores.*

El paso siguiente consiste en esquematizar la solución y transformar dicho esquema en un algoritmo que solucione el problema. Volveremos a emplear el diseño descendente con descomposición por refinamientos sucesivos, así entonces, el primer nivel del diseño es:

```
Calcular la media, la mediana y la moda para un conjunto de datos dado
```

De aquí procedemos al primer refinamiento de la solución, tomando en consideración que el cálculo de la mediana requiere que los datos se encuentren ordenados, así mismo, que se nos requiere imprimirlos en dicha alineación:

```
Inicializar variables
Solicitar datos de entrada
SI se introdujeron datos ENTONCES
    Ordenar los datos introducidos
    Calcular la media de los datos introducidos
    Calcular la mediana de los datos introducidos
```

```
    Calcular la moda de los datos introducidos
    Imprimir los datos introducidos y las medidas de tendencia central
SI NO
    Mostrar mensaje de error
FIN SI
```

Nótese que resulta importante comprobar si el usuario introdujo al menos un dato para los cálculos pretendidos, de lo contrario no será posible realizarlos.

En el segundo refinamiento manejamos ya variables. Requerimos una variable que almacene el conjunto de datos introducido, otra que indique cuántos datos se introdujeron y, tres más, para almacenar las estadísticas calculadas de la media, la mediana y la moda. Por tanto, la instrucción:

```
Inicializar variables
```

se puede refinar como sigue:

```
numDatos ← 0
(El conjunto de) datos ← ø (cero datos introducidos)
media ← 0
mediana ← 0
moda ← 0
```

Con la finalidad de estructurar apropiadamente el algoritmo de la solución, definiremos un módulo (función) para cada una de las tareas restantes que constituyen el primer refinamiento (y que más adelante refinaremos individualmente). Entonces, el módulo principal del algoritmo quedaría de la manera siguiente:

```
Calcular_estadísticas
    numDatos ← 0
    (El conjunto de) datos ← ø (cero datos introducidos)
    media ← 0
    mediana ← 0
    moda ← 0
```

```
    Solicitar_datos(datos, numDatos)

    SI numDatos > 0 ENTONCES
        Ordenar_datos(datos, numDatos)
        Calcular_media(datos, numDatos, media)
        Calcular_mediana(datos, numDatos, mediana)
        Calcular_moda(datos, numDatos, moda)

        Imprimir_datos(datos, numDatos)
        Mostrar la media, la mediana y la moda
    SI NO
        Mostrar "No se introdujeron datos."
    FIN SI
FIN
```

El algoritmo anterior ya no requiere más refinamiento como tal, pero sí lo requieren cada uno de los módulos que acabamos de introducir, entonces, comenzamos con el tercer refinamiento. Para el módulo `Solicitar_datos` requerimos una variable para recibir cada uno de los datos individuales introducidos por el usuario, con la cual comprobaremos si se introdujo el valor *centinela*.[i] El parámetro que el módulo recibe, `datos`, es la variable en la cual almacenaremos, como conjunto, cada uno de los valores introducidos. El otro parámetro, `numDatos`, almacenará el número total de datos que el usuario introdujo. Por tanto, el módulo queda así:

```
Solicitar_datos(datos, numDatos)
    datos ← ∅
    numDatos ← 0

    Solicitar un número entero y validar que sea ≤ 10

    MIENTRAS número introducido >= 0 HACER
```

[i] Un valor centinela es un valor especial que indica que la introducción de datos, en un programa, ha terminado. Esto es, mediante una estructura de control de repetición se reciben individualmente datos hasta el momento en el que se introduce el valor centinela. Por este motivo, el valor centinela nunca debe ser un valor que se pueda confundir con una entrada válida para el programa.

```
        // Agregar el número introducido al conjunto datos
        datos ← datos + número introducido
        numDatos ← numDatos + 1

        Solicitar un número entero y validar que sea ≤ 10
    FIN MIENTRAS
FIN
```

Continuando con el tercer refinamiento, tenemos que el módulo `Ordenar_datos` emplea el algoritmo de *ordenamiento por selección*,[5] para ordenar el conjunto de datos proporcionado. Este algoritmo funciona seleccionando el menor de los elementos del conjunto y colocándolo en primer lugar; a continuación selecciona el elemento menor siguiente y lo pone en el segundo lugar, y así sucesivamente, hasta terminar con el último de los elementos. En pseudocódigo esto luce así:

```
Ordenar_datos(datos, numDatos)
    PARA i ← 1 HASTA numDatos - 1 HACER
        minj ← i

        // datos(j) es el j-ésimo elemento de datos
        PARA j ← i + 1 HASTA numDatos HACER
            SI datos(j) < datos(minj) ENTONCES
                minj ← j
            FIN SI
        FIN PARA

        Intercambiar datos(i) y datos(minj)
    FIN PARA
FIN
```

El módulo siguiente que se desarrolla es `Calcular_media`, el cual requiere de una variable índice, que nos ayude a sumar cada uno de los elementos del conjunto, así como la propia variable que conserve la sumatoria, para luego calcular la media.

```
Calcular_media(datos, numDatos, media)
    suma ← 0
```

```
        PARA i ← 1 HASTA numDatos HACER
            suma ← suma + datos(i)
        FIN PARA

        media ← suma / numDatos
    FIN
```

El módulo `Calcular_mediana`, habiéndose ya ordenado ascendentemente los elementos del conjunto, consiste prácticamente en determinar si el número de elementos de este es par o impar ya que, si es impar, la mediana es el elemento central del conjunto, si no, será el promedio de los dos elementos centrales.

```
    Calcular_mediana(datos, numDatos, mediana)
        indiceMediana ← numDatos/2
        SI numDatos es impar ENTONCES
            mediana ← datos(indiceMediana + 1)
        SI NO
            mediana ← (datos(indiceMediana) + datos(indiceMediana + 1))/2
        FIN SI
    FIN
```

El cálculo de la tercera medida de tendencia central, la moda, requiere otra variable de conjunto que contabilice la frecuencia con que aparecen cada uno de los datos del 0 al 10, al momento de recorrer el conjunto de datos introducido por el usuario. Finalmente, el mayor de los elementos del conjunto de *frecuencias* señalará el valor que corresponde a la moda.

```
    Calcular_moda(datos, numDatos, moda)
        Inicializar el conjunto frecuencias con 11 elementos con valor 0

        PARA i ← 1 HASTA numDatos HACER
            // Los datos introducidos van de 0 a 10, pero el primer
elemento
            // de un conjunto es el elemento 1, no el elemento 0
            Sumar 1 al elemento de frecuencias indicado por datos(i)+1
        FIN PARA
```

```
    iMaximo ← 1

    PARA i ← 2 HASTA 11 HACER
        SI frecuencias(iMaximo) < frecuencias(i) ENTONCES
            iMaximo ← i
        FIN SI
    FIN PARA

    // De nuevo, considerar que el conjunto inicia en el elemento 1,
    // no el 0, pero los datos van de 0 a 10
    moda ← iMaximo - 1
FIN
```

Debemos observar los comentarios en el pseudocódigo anterior, esto es, no debemos pasar por alto que los datos que el usuario va a introducir pueden estar en el rango de 0 a 10, pero el acceso a los elementos de un conjunto, lo hacemos empezando con el elemento 1, no con el elemento 0, como lo hace el rango permitido. Entonces, debemos sumar un 1 a un valor del rango para convertirlo en índice de un conjunto y viceversa, debemos restar un 1 a un valor de índice de un conjunto para convertirlo a un valor del rango permitido de valores a introducir.

Por último, el módulo `Imprimir_datos` solamente presenta en pantalla los elementos de un conjunto:

```
Imprimir_datos(datos, numDatos)
    Mostrar "Elementos["
    PARA i ← 1 HASTA numDatos HACER
        Mostrar " ", datos(i)
    FIN PARA

    Mostrar " ]"
FIN
```

El algoritmo, con este tercer refinamiento, está casi completo; lo único que falta refinar, es la instrucción siguiente del módulo `Solicitar_datos`:

```
Solicitar un número entero y validar que sea ≤ 10
```

Como esta instrucción se emplea más de una vez en el módulo `Solicitar_datos`, conviene transformarla, a su vez, en un módulo (función) también. Así, el cuarto refinamiento del diseño de la solución consiste solamente en definir el módulo `Leer_dato` el cual, de acuerdo con el planteamiento del problema, debe recibir un dato entero entre 0 y 10, o terminar si el dato introducido es < 0. Por tanto, el pseudocódigo queda del modo siguiente:

```
Leer_dato(unDato)
    HACER
        Mostrar "Teclee un entero x, 0 <= x <= 10 (-1 para terminar):
"
        Leer unDato
    MIENTRAS unDato > 10
FIN
```

El diseño completo del algoritmo, incluyendo el cuarto refinamiento, queda como a continuación se presenta:

```
Calcular_estadísticas
    numDatos ← 0
    (El conjunto de) datos ← ø (cero datos introducidos)
    media ← 0
    mediana ← 0
    moda ← 0

    Solicitar_datos(datos, numDatos)

    SI numDatos > 0 ENTONCES
        Ordenar_datos(datos, numDatos)
        Calcular_media(datos, numDatos, media)
        Calcular_mediana(datos, numDatos, mediana)
        Calcular_moda(datos, numDatos, moda)

        Imprimir_datos(datos, numDatos)
        Mostrar la media, la mediana y la moda
```

```
        SI NO
            Mostrar "No se introdujeron datos."
        FIN SI
    FIN

    Solicitar_datos(datos, numDatos)
        datos ← ø
        numDatos ← 0

        Leer_dato(unDato)

        MIENTRAS unDato >= 0 HACER
            datos ← datos + unDato // Agregar unDato al conjunto datos
            numDatos ← numDatos + 1

            Leer_dato(unDato)
        FIN MIENTRAS
    FIN

    Leer_dato(unDato)
        HACER
            Mostrar "Teclee un entero x, 0 <= x <= 10 (-1 para terminar):
"
            Leer unDato
        MIENTRAS unDato > 10
    FIN

    Ordenar_datos(datos, numDatos)
        PARA i ← 1 HASTA numDatos - 1 HACER
            minj ← i

            // datos(j) es el j-ésimo elemento de datos
            PARA j ← i + 1 HASTA numDatos HACER
                SI datos(j) < datos(minj) ENTONCES
                    minj ← j
                FIN SI
            FIN PARA
```

```
            Intercambiar datos(i) y datos(minj)
        FIN PARA
    FIN

    Calcular_media(datos, numDatos, media)
        suma ← 0

        PARA i ← 1 HASTA numDatos HACER
            suma ← suma + datos(i)
        FIN PARA

        media ← suma / numDatos
    FIN

    Calcular_mediana(datos, numDatos, mediana)
        indiceMediana ← numDatos/2
        SI numDatos es impar ENTONCES
            mediana ← datos(indiceMediana + 1)
        SI NO
            mediana ← (datos(indiceMediana) + datos(indiceMediana + 1))/2
        FIN SI
    FIN

    Calcular_moda(datos, numDatos, moda)
        Inicializar el conjunto frecuencias con 11 elementos con valor 0

        PARA i ← 1 HASTA numDatos HACER
            // Los datos introducidos van de 0 a 10, pero el primer
elemento
            // de un conjunto es el elemento 1, no el elemento 0
            Sumar 1 al elemento de frecuencias indicado por (datos(i)+1)
        FIN PARA

        iMaximo ← 1

        PARA i ← 2 HASTA 11 HACER
            SI frecuencias(iMaximo) < frecuencias(i) ENTONCES
                iMaximo ← i
```

```
        FIN SI
    FIN PARA

    // De nuevo, considerar que el conjunto inicia en el elemento 1,
    // no el 0, pero los datos van de 0 a 10
    moda ← iMaximo - 1
FIN

Imprimir_datos(datos, numDatos)
    Mostrar "Elementos["
    PARA i ← 1 HASTA numDatos HACER
        Mostrar " ", datos(i)
    FIN PARA

    Mostrar " ]"
FIN
```

2.2.2 Beneficios del diseño modular

El empleo del diseño modular conlleva los beneficios siguientes:[6]

- *Facilidad de comprensión*: cada módulo/función debe realizar una (y solo una) tarea.
- *Reutilización de código*: los módulos/funciones de un algoritmo se pueden emplear también en otros algoritmos.
- *Eliminación de redundancia*: emplear módulos/funciones ayuda a evitar el reescribir el mismo segmento de código.
- *Eficiencia de mantenimiento*: cada módulo/función debe estar autocontenido, por lo tanto, no debe tener (o tener muy poco) efecto sobre otros módulos/funciones del algoritmo.

Si se emplean módulos durante el desarrollo del algoritmo es conveniente esquematizar la jerarquía de estos, a fin de clarificar las interacciones entre ellos. Para el algoritmo que acabamos de desarrollar, el diagrama de jerarquía de módulos

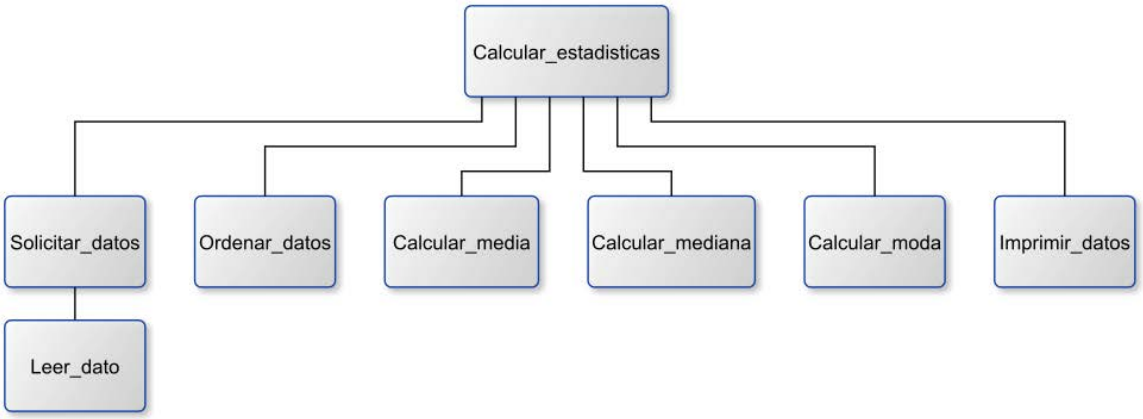

Figura 1. Jerarquía de módulos para el cálculo de las medidas de tendencia central

se presenta en la Figura 1.

Llegados a este punto del refinamiento, tenemos ya el detalle suficiente para pasar a la etapa siguiente del ciclo de desarrollo de software, probar que el algoritmo creado sea correcto. Como en el capítulo anterior, tomaremos algunos datos de entrada y calcularemos el resultado manualmente. Después, compararemos los resultados calculados contra los resultados que obtengamos de la ejecución de escritorio que hagamos del algoritmo, con los mismos datos de prueba.

Datos de entrada	3, 4, 2, 3, 2, 1, 1, 2, 1, 1, 2, 1, 1
Cálculo manual	Datos = [1 1 1 1 1 1 2 2 2 2 3 3 4] La media es: 1.85 La mediana es: 2.00 La moda es: 1
Ejecución de escritorio	Datos = [1 1 1 1 1 1 2 2 2 2 3 3 4] La media es: 1.85 La mediana es: 2.00 La moda es: 1

Haremos una segunda prueba para comprobar que la mediana se calcula de la manera esperada, cuando el número de datos introducidos es par:

Datos de entrada	3, 4, 3, 2, 1, 1, 2, 1, 1, 2, 1, 1
Cálculo manual	Datos = [1 1 1 1 1 1 2 2 2 3 3 4] La media es: 1.83 La mediana es: 1.50 La moda es: 1
Ejecución de escritorio	Datos = [1 1 1 1 1 1 2 2 2 3 3 4] La media es: 1.83 La mediana es: 1.50 La moda es: 1

Nuevamente no hemos incluido el detalle de la ejecución de escritorio debido a la restricción de espacio, pero dejamos esto como otro ejercicio al lector interesado en comprobar el funcionamiento pormenorizado de cada módulo del algoritmo desarrollado. Así también, se puede observar que los resultados del cálculo manual y los de la ejecución de escritorio son idénticos, lo que indica que, al menos para ambos conjuntos de valores de prueba, el algoritmo realiza la labor encomendada de manera correcta.

El paso siguiente del ciclo de desarrollo del software consiste en codificar el algoritmo desarrollado en un lenguaje de programación. Como ya sabemos, la codificación de nuestra solución al cálculo de las medidas de tendencia central la realizaremos en el lenguaje C++, cuyo código fuente se presenta a continuación:

```
 1.  // Listado 2: estadisticas.cpp
 2.  // Calcula la media, mediana y moda para un conjunto de datos dado.
 3.  //  Imprime el conjunto de datos ordenado y las estadísticas
     calculadas.
 4.  #include <iostream>
 5.  #include <iomanip>
 6.
 7.  using namespace std;
 8.
 9.  // Prototipos de función
10.  int solicitarDatos(int datos[]);
11.  int leerDato(void);
12.  void ordenarDatos(int datos[], int numDatos);
13.  void intercambiar(int& n1, int& n2);
14.  double calcularMedia(int datos[], int numDatos);
15.  double calcularMediana(int datos[], int numDatos);
```

```
16. int calcularModa(int datos[], int numDatos);
17. void imprimirDatos(int datos[], int numDatos);
18.
19. // Definición de constantes
20. const unsigned int MAX_DATOS = 25;
21.
22.
23. int main(void)
24. {
25.     int datos[MAX_DATOS] = {0};
26.     int numDatos = solicitarDatos(datos);
27.
28.     if (numDatos > 0)
29.     {
30.         ordenarDatos(datos, numDatos);
31.
32.         double media = calcularMedia(datos, numDatos);
33.         double mediana = calcularMediana(datos, numDatos);
34.         int moda = calcularModa(datos, numDatos);
35.
36.         imprimirDatos(datos, numDatos);
37.         cout << "\nLa media es: " << setprecision(2) << fixed <<
    media
38.              << "\nLa mediana es: " << mediana
39.              << "\nLa moda es: " << moda << endl;
40.     }
41.     else
42.     {
43.         cout << "\nNo se introdujeron datos." << endl;
44.     }
45.
46.     return 0;
47.
48. } // main()
49.
50.
51. // Solicita y lee datos enteros en el rango [0 - 10]. Un número
    negativo
52. // finaliza la entrada de datos.
53. // datos: Arreglo donde se recibirán los elementos solicitados.
54. // Devuelve el número de datos leido.
55. int solicitarDatos(int datos[])
56. {
57.     int numDatos = 0;
58.     int unDato = leerDato();
59.
60.     while (unDato >= 0)
61.     {
62.         datos[numDatos] = unDato;
63.         ++numDatos;
64.
65.         unDato = leerDato();
66.
67.     } // while
68.
69.     return numDatos;
```

```
70.
71.  } // solicitarDatos()
72.
73.
74.  // Lee un dato de tipo entero <= 10. Un número negativo
75.  // finaliza la entrada de datos.
76.  // Devuelve el dato leido.
77.  int leerDato(void)
78.  {
79.      int unDato = 0;
80.
81.      do
82.      {
83.          cout << "\nTeclee un entero x, 0 ≤ x ≤ 10(-1 para terminar): ";
84.          cin >> unDato;
85.
86.      } while (unDato > 10);
87.
88.      return unDato;
89.
90.  } // leerDato()
91.
92.
93.  // Ordena los 'numDatos' primeros elementos de un arreglo
94.  // datos: Arreglo a ordenar
95.  // numDatos: Número de elementos a ordenar del arreglo
96.  void ordenarDatos(int datos[], int numDatos)
97.  {
98.      for (int i = 0; i < (numDatos - 1); ++i)
99.      {
100.         int minj = i;
101.
102.         for (int j = i + 1; j < numDatos; ++j)
103.         {
104.             if (datos[j] < datos[minj])
105.             {
106.                 minj = j;
107.             }
108.
109.         } // for j
110.
111.         intercambiar(datos[i], datos[minj]);
112.
113.     } // for i
114.
115. } // ordenarDatos()
116.
117.
118. // Intercambia el valor de dos variables
119. // n1: Primera variable a intercambiar
120. // n2: Segunda variable a intercambiar
121. void intercambiar(int& n1, int& n2)
122. {
123.     int temp = n1;
124.     n1 = n2;
125.     n2 = temp;
```

```
126.
127.  } // intercambiar()
128.
129.
130.  // Calcula la media de los elementos de un arreglo
131.  // datos: Arreglo de elementos cuya media se va a calcular
132.  // numDatos: Número de elementos del arreglo cuya media se calculará
133.  double calcularMedia(int datos[], int numDatos)
134.  {
135.      double suma = 0.0;
136.
137.      for (int i = 0; i < numDatos; ++i)
138.      {
139.          suma += datos[i];
140.
141.      } // for i
142.
143.      return (suma / numDatos);
144.
145.  } // calcularMedia()
146.
147.
148.  // Calcula la mediana de los elementos de un arreglo
149.  // datos: Arreglo de elementos cuya mediana se va a calcular
150.  //  numDatos:  Número  de  elementos  del  arreglo  cuya  mediana  se
      calculará
151.  double calcularMediana(int datos[], int numDatos)
152.  {
153.      double mediana = 0.0;
154.      int indiceMediana = numDatos / 2;
155.
156.      if ((numDatos % 2) == 1) // El número de datos es impar
157.      {
158.          mediana = datos[indiceMediana];
159.      }
160.      else // El número de datos es par
161.      {
162.          double suma = static_cast<double>(datos[indiceMediana - 1] +
163.                                            datos[indiceMediana]);
164.          mediana = suma / 2;
165.      }
166.
167.      return mediana;
168.
169.  } // calcularMediana()
170.
171.
172.  // Calcula la moda de los elementos de un arreglo
173.  // datos: Arreglo de elementos cuya moda se va a calcular
174.  // numDatos: Número de elementos del arreglo cuya moda se calculará
175.  int calcularModa(int datos[], int numDatos)
176.  {
177.      int frecuencias[11] = {0};
178.
179.      // Contar las ocurrencias de cada valor
180.      for (int i = 0; i < numDatos; ++i)
```

```
181.      {
182.          ++frecuencias[ datos[i] ];
183.
184.      } // for i
185.
186.      int iMaximo = 0;
187.
188.      // Buscar la moda
189.      for (int i = iMaximo + 1; i < 11; ++i)
190.      {
191.          if (frecuencias[iMaximo] < frecuencias[i])
192.          {
193.              iMaximo = i;
194.          }
195.
196.      } // for i
197.
198.      return iMaximo;
199.
200.  } // calcularModa()
201.
202.
203.  // Imprime en pantalla los elementos de un arreglo
204.  // datos: Arreglo a imprimir
205.  // numDatos: Número de elementos a imprimir del arreglo
206.  void imprimirDatos(int datos[], int numDatos)
207.  {
208.      cout << "\nDatos = [";
209.
210.      for (int i = 0; i < numDatos; ++i)
211.      {
212.          cout << " " << datos[i];
213.
214.      } // for i
215.
216.      cout << " ]" << endl;
217.
218.  } // imprimirDatos()
```

2.2.3 La declaración *using*

La primera de las líneas del Listado 2 que no nos es familiar es la número 7:

```
using namespace std;
```

En el capítulo anterior hablábamos de los espacios de nombres y hacíamos la analogía con el apellido de una familia. En un entorno profesional de desarrollo es,

en cierto sentido, hasta obligatorio el calificar **cada** identificador con el operador de resolución de ámbito "`::`" y un espacio de nombres, es decir, ponerle *apellido* al identificador. En entornos de prueba o de enseñanza, como es nuestro caso, es posible emplear un atajo que nos ahorre el tener que colocar el calificador de nombre a cada identificador de la biblioteca estándar (o de otro espacio de nombres). La directiva `using namespace ns;` hace que todas las definiciones en el espacio de nombres *`ns`* estén disponibles para nuestro código. Con la línea 7 del Listado 2, le estamos diciendo a C++ que ponga a nuestra disposición todas las definiciones en el espacio de nombres `std`, esto es, que todos los nombres en nuestro código fuente, los considere como de apellido `std`.

Lo anterior puede parecer un ahorro en la escritura de código pero, al incluir *todas* las definiciones del espacio de nombres `std`, podemos volver al problema de colisión de nombres que esta característica de C++ (espacios de nombres) intenta resolver.

2.2.4 Declaración, definición y prototipos de función

En C++ las funciones deben ser conocidas antes de su primer uso. En el Listado 2, la función `main` hace uso de la función `solicitarDatos`, por lo tanto, esta debe ser conocida antes de que se utilice. Para resolver este problema en el ejemplo del Listado 2, podríamos poner la definición de la función `solicitarDatos` antes de la definición de la función `main`, lo cual resolvería la cuestión planteada.[ii] El problema difícil viene cuando tenemos una función `evaluaExpresion` que hace uso de una función `evaluaSubexpresion` la cual, a su vez, emplea a la propia función `evaluaExpresion`. Por ejemplo, si tenemos `(1 * 2) + (3 - 3)`,

[ii] La definición de una función consiste del código fuente que la implementa, es decir, el encabezado de la función más el conjunto de instrucciones que conforman su cuerpo, delimitado por las llaves de apertura ("{") y cierre ("}").

llamaremos a `evaluaExpresion` para evaluar la expresión completa. Al encontrar la subexpresión `(1 * 2)`, esta función llamará a `evaluaSubexpresion` para que calcule su valor. Cuando `evaluaSubexpresion` encuentra que dentro de los paréntesis existe una expresión que, por sí sola, es una expresión completa; llamará a `evaluaExpresion` para procesarla. El código fuente de este proceso luciría de forma similar al del Listado 3.

```
 1.  // Listado 3. sinprototipos.cpp
 2.  // Ejemplifica la necesidad del uso de prototipos de función.
 3.  …
 4.  int evaluaSubexpresion(string expresion)
 5.  {
 6.      …
 7.      evaluaExpresion(expresion);
 8.      …
 9.  }
10.
11.  int evaluaExpresion(string expresion)
12.  {
13.      …
14.      evaluaSubexpresion(expresion);
15.      …
16.  }
17.
18.  int main(void)
19.  {
20.      …
21.      string expresion = "(1 * 2) + (3 - 3)";
22.
23.      evaluaExpresion(expresion);
24.      …
25.  }
```

Cada función debe ser conocida antes de su primer uso, entonces, si aplicamos la solución anteriormente descrita y ponemos la definición de la función `evaluaSubexpresion` antes de la definición de `evaluaExpresion`, tal y como muestra el Listado 3, la llamada que realiza `main` en la línea 23 es correcta, puesto que `evaluaExpresion` es conocida en este punto del código. Luego, la llamada que realiza `evaluaExpresion` en la línea 14 también es correcta, puesto que `evaluaSubexpresion` ya es conocida en este punto del código fuente. El inconveniente viene con la llamada que realiza la función `evaluaSubexpresion`

en la línea 7, ya que en este punto del código la función `evaluaExpresion` aún no es conocida.

Podríamos invertir el orden de definición de las funciones `evaluaExpresion` y `evaluaSubexpresion` en el Listado 3, pero llegaríamos al mismo inconveniente, una función que no es conocida al momento de utilizarla por primera vez.

La forma de solucionar este problema consiste en *declarar* la función antes de su primer uso. Esto es, colocar solamente el encabezado de la función (tipo del valor que devuelve, nombre de la función y tipos de los parámetros que recibe) pero sin cuerpo, además de un punto y coma. A esta declaración se le denomina *prototipo* de la función y, como se acaba de señalar, no requiere la colocación de los nombres de los parámetros que la función recibe. Sin embargo, son útiles como documentación de la función, por lo que se acostumbra incluirlos. Entonces, el cambio requerido para el Listado 3, se muestra en el Listado 4.

```
 1.  // Listado 4. conprototipos.cpp
 2.  // Ejemplifica el uso de prototipos de función.
 3.  …
 4.  int evaluaExpresion(string expresion);
 5.  int evaluaSubexpresion(string expresion);
 6.
 7.  int main(void)
 8.  {
 9.      …
10.      string expresion = "(1 * 2) + (3 - 3)";
11.
12.      evaluaExpresion(expresion);
13.      …
14.  }
15.
16.  int evaluaExpresion(string expresion)
17.  {
18.      …
19.      evaluaSubexpresion(expresion);
20.      …
21.  }
22.
23.  int evaluaSubexpresion(string expresion)
24.  {
25.      …
26.      evaluaExpresion(expresion);
27.      …
```

```
28.  }
```

Volviendo al Listado 2, las líneas 10 a la 17 declaran los prototipos de las funciones que el programa requiere y cuya definición se proporciona más adelante.

2.2.5 Constantes con nombres

La definición de *número mágicos* en el lenguaje de programación C se realiza generalmente mediante la definición de una macro, es decir, empleando la directiva `#define` del preprocesador. Estos números mágicos que aparecen en casi cualquier programa, por ejemplo el número de elementos máximo que podemos recibir para el cálculo de las medidas de tendencia central en el programa del Listado 2, por sí mismos no conllevan significado alguno. Esto es, al colocar simplemente un número como `18` para limitar la edad de un individuo, no conlleva información significativa. Sin embargo, cuando empleamos en C, o también en C++, una línea como la siguiente:

```
#define MAYORIA_DE_EDAD 18
```

y la empleamos en una estructura condicional como esta:

```
if (edad >= MAYORIA_DE_EDAD)
{
    …
}
else
{
    …
}
```

entonces sí que podemos interpretar correctamente la información.[iii] El problema con este tipo de macros es que, a pesar de proporcionar una semántica mejorada -

[iii] La directiva `#define` *`TEXTO texto_de_reemplazo`*, le indica al preprocesador del lenguaje C y C++ que, en cualquier lugar del código fuente donde encuentre la cadena *`TEXTO`* –exceptuando

entendemos mejor lo que se pretende con el código-, al ser un simple reemplazo de un texto por otro, el compilador no puede realizar verificación de tipos. C++ va un paso más allá y nos permite definir *constantes con nombre*. En el Listado 2, la línea:

```
20. const unsigned int MAX_DATOS = 25;
```

define la constante con nombre `MAX_DATOS`, la cual permite no solo proporcionar un mejor significado sino, también, proporcionar al compilador la oportunidad de verificar que dicha constante sea empleada correctamente. Esto es, que la expresión en la que se emplee la constante, sea una expresión correcta del tipo `unsigned int`. Al igual que con la directiva `#define`, es costumbre escribir los nombres de constantes con todas sus letras en mayúsculas, así como separar las pablabras con caracteres de subrayado (a fin de direnciarlas rápidamente de los nombres de las variables).

2.2.6 Arreglos

En el programa de cálculo de estadísticas, empleamos una *estructura de datos* de tipo *arreglo* para almacenar los valores introducidos por el usuario.[7] Una estructura de datos es una colección de elementos, generalmente de alguno o varios de los tipos básicos, que tiene como finalidad facilitar el trabajo con dichos elementos.

La más sencilla de las estructuras de datos es el arreglo, al que también se le conoce como *vector*, *lista*, por su nombre en inglés *array* o, cuando el arreglo tiene más de una dimensión, *matriz*. El arreglo es una colección de un número fijo de elementos del mismo tipo de datos, colocados en fila cuando es de una dimensión y colocados

---

dentro de una cadena colocada entre comillas-, la reemplace con la cadena `texto_de_reemplazo`. Así, el compilador nunca recibe la cadena *`TEXTO`* sino *`texto_de_reemplazo`*, y ésta es la que formará parte del proceso de compilación y, por lo tanto, del programa ejecutable generado.

en filas y columnas cuando es de dos dimensiones. La línea 25 del Listado 2 declara e inicializa un arreglo de `MAX_DATOS` elementos, todos inicializados en 0. La forma:

```
int datos[10] = {0};
```

es una forma corta de inicialización equivalente a:

```
int datos[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

El acceso a los elementos de un arreglo se realiza indicando la posición, *índice* o *subíndice* del elemento en el arreglo. Como ya se sabe, los índices de arreglos en C y C++ inician en 0, esto es, el primer elemento de un arreglo `a` es `a[0]` y, el último, es `a[`*`numero_de_elementos_en_el_arreglo`*` - 1]`.

La manipulación de los arreglos se realiza generalmente por medio de estructuras de control de iteración `for`, dado que de antemano se conoce el número de elementos de un arreglo, por lo que dicha estructura resulta ideal. Cuando el arreglo es una matriz (un arreglo de dos dimensiones) el acceso se realiza indicando dos índices (por ejemplo `a[2][2]`); cuando el arreglo es de *n* dimensiones, el acceso se realiza indicando *n* índices (`a[2]…[2]` -el `[2]` se repite *n* veces-). Análogamente, la manipulación (inicialización, lectura, modificación, etc.) de todos los elementos de un arreglo, ocupa un solo ciclo `for`; el de una matriz ocupa dos ciclos `for`; y, el de *n* dimensiones, ocupará *n* ciclos `for`.

2.2.7 Interacción entre funciones

Al emplear la modularización de la solución de un problema, debemos considerar también la interacción entre los módulos, entre menor y mejor definida sea la comunicación, mejor será el diseño de nuestra solución. Esto es, al diseñar los módulos que compondrán un programa, debemos evitar en la medida de lo posible el *acoplamiento* entre ellos (la dependencia de unos con otros); entre más independiente sea cada módulo, mayor será la posibilidad que tengamos para

reutilizarlo en otro u otros algoritmos, aunado a que su mantenimiento será más fácil. La interacción entre los módulos se puede realizar a través de la manipulación de datos globales, locales y/o a través del paso de parámetros a y desde cada módulo.

2.2.7.1 Ámbito de una variable: variables globales y locales[1,2,3,6,7]

El *ámbito* de una variable es la zona del programa en donde la variable se encuentra *viva*, es decir, donde ha sido definida y en la cual es posible hacer referencia a ella. Con esto en mente, las variables pueden ser *globales*, esto es, el ámbito de la variable consiste de todo el programa o, en otras palabras, puede ser utilizada por todos los módulos/funciones, lo cual es su mayor ventaja porque facilita mucho su manejo, pero también su mayor desventaja, ya que cualquier función puede alterar el valor de esta variable aún inadvertidamente, error que es muy difícil de depurar. Además, una variable global se puede acceder desde que el programa inicia hasta que acaba.

La otra forma de variables son las *locales*, el ámbito de una variable local es solamente la función en donde se encuentra definida, empiezan a vivir cuando la función es llamada y mueren cuando la función termina. Este tipo de variable evita el acoplamiento entre funciones y los efectos colaterales, como la modificación por error de una variable global. En la medida de lo posible, es el de tipo de variable que debemos emplear, exceptuando al momento de enviar y recibir información hacia una función, en ese caso debemos emplear el paso de parámetros y los valores de devolución.

2.2.7.2 Paso de parámetros: parámetros formales y parámetros reales. Valores de devolución [1,2,3,6,7,8]

En la introducción del capítulo se menciona que cuando una función llama o emplea los servicios de otra, la función que llama puede enviar información a la función llamada así como recibir, también, un valor devuelto por esta. La función que requiera información para realizar su tarea (o que devuelva un valor), debe especificarlo en su propio encabezado, como una lista de datos a los que se les denomina *parámetros formales* o, simplemente, *parámetros* (una función solo puede devolver *un* valor en específico a la función que la llama, por lo que no requiere una lista, aunque este valor puede ser una estructura de datos constituida por más de un valor).

En C++ la lista de parámetros formales incluye el tipo de cada dato requerido así como la asignación de un nombre, esto porque los valores que la función recibe en realidad actúan como variables locales,[iv] las cuales se pueden manipular dentro de la función (lo cual no se aconseja) cuando esta esté en ejecución. Estos valores, que son los que en realidad se pasan a la función, reciben el nombre de *parámetros reales* o *argumentos*. Por ejemplo, en la línea 111 del Listado 2, los elementos `datos[i]` y `datos[minj]` son los parámetros reales para la función `intercambiar`; mientras que, en la línea 121, los parámetros formales de esta función son `n1` y `n2`.

En C++ el valor devuelto por una función también se indica en el encabezado de la función, pero antes del nombre de esta. Sin embargo, solo se indica el tipo del dato que se devuelve ya que, en este contexto, el nombre no es de utilidad porque su ámbito finalizaría al mismo tiempo que la función finaliza.

[iv] De ahí que también se les denomine *variables de parámetros*.

Continuando con el análisis, en la línea 26 del Listado 2 la función principal llama a la función `solicitarDatos`, a fin de requerir al usuario del programa que ingrese los datos a procesar. Esta función devuelve un valor de tipo entero, indicando el número real de datos ingresados, si este número es igual a 0 no hay datos que procesar, por lo que el programa termina presentando un mensaje al respecto (líneas 41 a 48). Si el número de datos ingresado es de al menos 1, se llama a la función `ordenarDatos` para que ordene ascendentemente el arreglo con los datos ingresados (línea 30).

Después de ordenar los datos, en las líneas 31 a 35 la función `main` llama a las funciones correspondientes al cálculo de cada medida de tendencia central (`calcularMedia`, `calcularMediana` y `calcularModa`), las cuales devuelven el valor respectivo a su cálculo. Las dos primeras funciones devuelven un valor de tipo `double`, ya que `calcularMedia` realiza una división cuyo resultado, generalmente, no será un número entero; y, `calcularMediana`, debido a que si el número de elementos en el arreglo `datos` es par, se tendrán que promediar los dos valores centrales del arreglo. `calcularModa` devuelve el valor que más se repite en el arreglo `datos`, por tanto, devuelve un valor de tipo entero.

2.2.7.3 Formato de números de punto flotante

En el Listado 2, las líneas 37 a 39:

```
cout << "\nLa media es: " << setprecision(2) << fixed << media
     << "\nLa mediana es: " << mediana
     << "\nLa moda es: " << moda << endl;
```

emplean el *manipulador de flujo parametrizado* `setprecision`. Ya se había comentado acerca del manipulador de flujo *no parametrizado* `endl`[v] y cómo afecta su uso la salida impresa en la pantalla. El manipulador `setprecision(n)`, indica que a partir de ese momento los números de punto flotante que se envíen al flujo de salida que lo emplea, deberán mostrarse con *n* dígitos de precisión (dígitos después del punto decimal) empleando redondeo si es necesario. Este manipulador de flujo requiere la inclusión del archivo de cabecera `iomanip`. De la misma manera, `fixed` es otro manipulador de flujo que indica que todos los valores de punto flotante que se envíen al flujo de salida que lo emplea, deberán mostrarse en notación de punto fijo, esto es, mostrando forzosamente un número específico de dígitos, además del punto decimal y los ceros necesarios para alcanzar ese número de dígitos.

Los manipuladores de flujo no parametrizados `endl` y `fixed` se ubican en el archivo de cabecera `iostream` por lo que no requieren la inclusión del archivo `iomanip`.

Prosiguiendo con el estudio del Listado 2, la función `solicitarDatos` (líneas 51 a 71) llama a la función `leerDato` (líneas 74 a 90 –observe el uso de la estructura de control `do…while` en esta función-) para validar que los números ingresados se encuentren en el rango 0 a 10, además de finalizar la entrada con el primer valor negativo que se ingrese. De acuerdo con las especificaciones en el planteamiento del problema, es responsabilidad total del usuario el ingresar como máximo `MAX_DATOS` (25) números, por lo tanto, esta comprobación no se realiza (como ejercicio de práctica, el lector puede añadir la codificación de esta otra validación).

[v] `setprecision` es un manipulador de flujo *parametrizado*, debido a que necesita un número –un parámetro- para realizar su tarea; mientras que `endl` no lo requiere, por lo cual se le denomina *no parametrizado*.

2.2.7.4 Paso de parámetros por valor y por referencia

Cuando un dato de alguno de los tipos fundamentales se pasa a una función, C++ primero crea la variable correspondiente al parámetro formal, luego realiza una copia del valor del parámetro real y la asigna al parámetro formal. Esto se denomina *paso de parámetros por valor* y sirve para proteger de modificaciones accidentales a la variable original (el parámetro real) en la función que llama. Aunque esto es lo deseable, en ciertas circunstancias no es lo que se pretende. Por ejemplo, la función `ordenarDatos` (líneas 93 a 115 del Listado 2) procesa todos los elementos del arreglo `datos`, ordenándolos de menor a mayor. Para tal efecto se apoya de la función `intercambiar` (líneas 118 a 127), cuyo objetivo es colocar el valor de su primer argumento en el segundo, y colocar el valor de su segundo argumento en el primero. La primera solución que se nos viene a la mente es codificar esta función como se ejemplifica en el Listado 5. Además, la salida del programa es la siguiente:

```
 1.  // Listado 5: pasoporvalor.cpp
 2.  // Demuestra el paso por valor de argumentos a funciones.
 3.  #include <iostream>
 4.  #include <string>
 5.
 6.  using namespace std;
 7.
 8.  // Prototipos de función
 9.  void imprimirValores(string mensaje, int var1, int var2);
10.  void intercambiar(int n1, int n2);
11.
12.  int main(void)
13.  {
14.      int valor1 = 15;
15.      int valor2 = 20;
16.
17.      imprimirValores("- main(): Antes del intercambio -",
18.                      valor1, valor2);
19.
20.      intercambiar(valor1, valor2);
21.
22.      imprimirValores("- main(): Realizado el intercambio -",
23.                      valor1, valor2);
24.
25.      return 0;
26.
27.  } // main()
28.
29.
```

```
30.  // Imprime en pantalla el valor de dos variables enteras
31.  // mensaje: Mensaje a imprimir antes de los valores de las variables
32.  // var1: Primera variable a imprimir
33.  // var2: Segunda variable a imprimir
34.  void imprimirValores(string mensaje, int var1, int var2)
35.  {
36.      cout << "\n" << mensaje;
37.      cout << "\nLa primera variable tiene el valor: " << var1
38.           << "\nLa segunda variable tiene el valor: " << var2
39.           << endl;
40.
41.  } // imprimirValores()
42.
43.
44.  // Intercambia el valor de dos variables
45.  // n1: Primera variable a intercambiar
46.  // n2: Segunda variable a intercambiar
47.  void intercambiar(int n1, int n2)
48.  {
49.      imprimirValores("- intercambiar(): Antes del intercambio -",
50.                      n1, n2);
51.
52.      int temp = n1;
53.      n1 = n2;
54.      n2 = temp;
55.
56.      imprimirValores("- intercambiar(): Realizado el intercambio -",
57.                      n1, n2);
58.
59.  } // intercambiar()
```

```
- main(): Antes del intercambio -
La primera variable tiene el valor: 15
La segunda variable tiene el valor: 20

- intercambiar(): Antes del intercambio -
La primera variable tiene el valor: 15
La segunda variable tiene el valor: 20

- intercambiar(): Realizado el intercambio -
La primera variable tiene el valor: 20
La segunda variable tiene el valor: 15

- main(): Realizado el intercambio -
La primera variable tiene el valor: 15
La segunda variable tiene el valor: 20
```

Como podemos observar, las variables originales no han cambiado. Como ya se había comentado, esto se debe a que C++ copia el valor de los parámetros reales a los parámetros formales al momento de realizar la llamada a la función `intercambiar`, cualquier modificación que se realice a estos parámetros formales no tendrá efecto en el valor de los parámetros reales.

Si el objetivo es modificar el valor de los parámetros reales, las variables originales, debemos emplear el paso por referencia.

2.2.8 Paso por referencia y parámetros de referencia[1,2,3,7,8]

Ya vimos que el paso por valor tiene el efecto (a veces bueno, a veces malo) de no modificar el valor de los parámetros reales. Otro inconveniente es que la copia que se realiza puede llegar a consumir un tiempo considerable si se pasa como parámetro de una función, por ejemplo, una variable de estructura o un objeto (que se tratará en los capítulos siguientes) muy grande, o también, si la llamada a la función (y, por ende, la copia de un valor de un tipo básico) se realiza dentro de un ciclo que se repite una gran cantidad de ocasiones.

Para resolver los dos problemas anteriores, el paso de parámetros *por referencia* emplea *variables de apuntador*. Estas variables cumplen con todas las características de cualquier variable normal, la diferencia consiste en que en lugar de mantener un valor en específico, mantienen la dirección en la memoria de la computadora en donde se ubica otra variable.[vi] Así, cuando se pasa un apuntador como parámetro a una función, solo se pasa este y no una copia completa de la variable original (cuya dirección guarda el apuntador, es decir, la variable a la que *apunta* el apuntador). También se resuelve la necesidad de modificar el dato original, como el apuntador

[vi] De ahí el nombre de paso de parámetros por referencia, es decir, lo que se pasa a la función es una referencia de la ubicación del parámetro real.

señala la ubicación de este, podemos acceder a él sin problemas; para consultar su valor o para modificarlo.

Desafortunadamente el uso de apuntadores siempre ha correlacionado con dificultades conceptuales para su entendimiento y aplicación, por tal motivo algunos lenguajes, por ejemplo Java, han suprimido su empleo explícito. C++ no se ha quedado atrás y ha implementado los *parámetros de referencia*.[9] Un parámetro de referencia es un alias del argumento (parámetro real) correspondiente en la llamada a una función, por tanto, con un parámetro de referencia es posible modificar el valor original del argumento. La sintaxis de un parámetro de referencia consiste en agregar a la declaración (el prototipo) de la función y al encabezado de su definición, un carácter "**&**" después del tipo de dato del parámetro de referencia. Luego, el manejo interno en la función de dicha variable es idéntico al de cualquier otra variable. Por ejemplo, haciendo las modificaciones pertinentes al Listado 5, tenemos ahora el programa del Listado 6, que intercambia de manera correcta los valores enviados a la función `intercambiar`.

```
 1.  // Listado 6: pasoporreferencia.cpp
 2.  // Demuestra el paso por referencia de argumentos a funciones
 3.  // (mediante parámetros de referencia).
 4.  #include <iostream>
 5.  #include <string>
 6.
 7.  using namespace std;
 8.
 9.  // Prototipos de función
10.  void imprimirValores(string mensaje, int var1, int var2);
11.  void intercambiar(int& n1, int& n2);
12.
13.  int main(void)
14.  {
15.      int valor1 = 15;
16.      int valor2 = 20;
17.
18.      imprimirValores("- main(): Antes del intercambio -",
19.                      valor1, valor2);
20.
21.      intercambiar(valor1, valor2);
22.
23.      imprimirValores("- main(): Realizado el intercambio -",
24.                      valor1, valor2);
25.
```

```
26.      return 0;
27.
28.  } // main()
29.
30.
31.  // Imprime en pantalla el valor de dos variables enteras
32.  // mensaje: Mensaje a imprimir antes de los valores de las variables
33.  // var1: Primera variable a imprimir
34.  // var2: Segunda variable a imprimir
35.  void imprimirValores(string mensaje, int var1, int var2)
36.  {
37.      cout << "\n" << mensaje;
38.      cout << "\nLa primera variable tiene el valor: " << var1
39.           << "\nLa segunda variable tiene el valor: " << var2
40.           << endl;
41.
42.  } // imprimirValores()
43.
44.
45.  // Intercambia el valor de dos variables
46.  // n1: Primera variable a intercambiar
47.  // n2: Segunda variable a intercambiar
48.  void intercambiar(int& n1, int& n2)
49.  {
50.      imprimirValores("- intercambiar(): Antes del intercambio -",
51.                      n1, n2);
52.
53.      int temp = n1;
54.      n1 = n2;
55.      n2 = temp;
56.
57.      imprimirValores("- intercambiar(): Realizado el intercambio -",
58.                      n1, n2);
59.
60.  } // intercambiar()
```

Y la salida del programa es la siguiente:

```
- main(): Antes del intercambio -
La primera variable tiene el valor: 15
La segunda variable tiene el valor: 20

- intercambiar(): Antes del intercambio -
La primera variable tiene el valor: 15
La segunda variable tiene el valor: 20

- intercambiar(): Realizado el intercambio -
La primera variable tiene el valor: 20
```

```
La segunda variable tiene el valor: 15

- main(): Realizado el intercambio -
La primera variable tiene el valor: 20
La segunda variable tiene el valor: 15
```

Como podemos observar, los únicos cambios necesarios al código fuente en el Listado 5, para lograr el objetivo pretendido del intercambio, consistieron en agregar los caracteres "`&`" en las líneas 11 y 48 del código fuente en el Listado 6.

2.2.9 Paso de parámetros de tipo arreglo

Las funciones `solicitarDatos`, `ordenarDatos`, `calcularMedia`, `calcularMediana`, `calcularModa` e `imprimirDatos` del Listado 2, reciben como parámetro el arreglo `datos` en el que se guardarán/modificarán o del que se leerán los datos introducidos por el usuario.

A primera vista pareciera que siendo un parámetro que es una colección de datos de tipo básico, debiera aplicársele el paso de parámetros por valor, lo que implicaría que las funciones que alteran el contenido del arreglo lo harían en una copia de este, no en el original y, por lo tanto, que dichas funciones no cumplen con su objetivo. La realidad es que, en virtud de que el arreglo es una colección de datos cuya copia pudiera consumir un tiempo excesivo, C++ aplica por defecto a los arreglos el paso de parámetros por referencia. Esto es, el nombre del arreglo realmente es un apuntador a su primer elemento, además de que el resto de elementos siempre se almacenan consecutivamente al primero. Así pues, conociendo la ubicación en memoria del primer elemento, podemos conocer la ubicación de todos los demás. Por este motivo, en el encabezado (y el prototipo) de la función que recibe como parámetro un arreglo, no es necesario indicar el número de elementos que contiene, solo se indican los corchetes "`[]`". La única diferencia entre el nombre de un arreglo

y un apuntador a su primer elemento, es la notación de acceso a través de estos corchetes.

Volviendo al código fuente del Listado 2, ya mencionamos que la función `ordenarDatos` (líneas 93 a 115) procesa todos los elementos del arreglo `datos`, ordenándolos de menor a mayor y que, para tal efecto, se apoya en la función `intercambiar` (líneas 118 a 127), misma que emplea parámetros de referencia. El algoritmo que esta función emplea para ordenar el arreglo, el ordenamiento por selección, es un método conocido y estudiado en el área de las ciencias computacionales, por lo que se puede consultar todo a su respecto en cualquier libro que trate sobre algoritmos en esta área.

En las líneas 130 a 145, 148 a 169 y 172 a 200, se encuentra el código de las funciones `calcularMedia`, `calcularMediana` y `calcularModa`, respectivamente, las cuales realizan el cálculo de la medida de tendencia central correspondiente. Las dos primeras se explican por sí solas, el cálculo de la moda consiste de dos pasos: en el primero se ocupa un arreglo de once elementos, los necesarios para almacenar la frecuencia de cada uno de los valores que el usuario puede introducir al programa (0, 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10) y, conforme se recorren todos los elementos del arreglo de los datos introducidos por el usuario, se van incrementando las veces que aparece cada uno de ellos (en el arreglo de frecuencias); el segundo paso consiste en encontrar en el arreglo de frecuencias cuál es la mayor, dicha frecuencia señala al valor entre 0 y 10 que es la moda.

Finalmente, en las líneas 203 a la 218 del Listado 2, se encuentra el código que imprime los elementos de un arreglo, la función `imprimirDatos`.

2.2.10 Sobrecarga de funciones

Antes de finalizar este capítulo y entrar de lleno en el mundo de la programación orientada a objetos, cabe mencionar otra característica de C++ que facilita el trabajo con funciones, concepto al que se le denomina *sobrecarga de funciones*.

En el lenguaje de programación C, si queremos realizar una operación sobre diferentes tipos de datos debemos duplicar la funcionalidad requerida, cuidando de no duplicar los nombres de las funciones que intervienen. Por ejemplo, en el Listado 7 se muestra la implementación de la operación suma para diferentes tipos de datos:

```
 1.  // Listado 7: sobrecarga.c
 2.  // Demuestra la forma de implementar la sobrecarga de funciones
 3.  // en el lenguaje de programación C
 4.  #include <stdio.h>
 5.  #include <stdlib.h>
 6.
 7.  // Prototipos de función
 8.  int sumaInt(int a, int b);
 9.  float sumaFloat(float a, float b);
10.  double sumaDouble(double a, double b);
11.
12.  int main()
13.  {
14.      int i1 = 10;
15.      int i2 = 20;
16.      float f1 = 10.0F;
17.      float f2 = 20.0F;
18.      double d1 = 10.0;
19.      double d2 = 20.0;
20.
21.      printf("\n- Suma int -\nLa suma de %d y %d es: %d\n",
22.             i1, i2, sumaInt(i1, i2));
23.      printf("\n- Suma float -\nLa suma de %.2f y %.2f es: %.2f\n",
24.             f1, f2, sumaFloat(f1, f2));
25.      printf("\n- Suma double -\nLa suma de %.2lf y %.2lf es:
     %.2lf\n",
26.             d1, d2, sumaDouble(d1, d2));
27.
28.      return 0;
29.
30.  } // main()
31.
32.
33.  // Suma dos números enteros
34.  // a: Primer número a sumar
```

```
35.  // b: Segundo número a sumar
36.  int sumaInt(int a, int b)
37.  {
38.      return (a + b);
39.
40.  } // sumaInt()
41.
42.
43.  // Suma dos números de tipo float
44.  // a: Primer número a sumar
45.  // b: Segundo número a sumar
46.  float sumaFloat(float a, float b)
47.  {
48.      return (a + b);
49.
50.  } // sumaFloat()
51.
52.
53.  // Suma dos números de tipo double
54.  // a: Primer número a sumar
55.  // b: Segundo número a sumar
56.  double sumaDouble(double a, double b)
57.  {
58.      return (a + b);
59.
60.  } // sumaDouble()
```

Cuya salida, como es de esperar, es la siguiente:

```
- Suma int -
La suma de 10 y 20 es: 30


- Suma float -
La suma de 10.00 y 20.00 es: 30.00


- Suma double -
La suma de 10.00 y 20.00 es: 30.00
```

Lo complicado de esta implementación es que debemos recordar el nombre dado a la función para cada tipo de datos: `sumaInt`, `sumaFloat` y `sumaDouble`. Más aún si quisiéramos que cada una de estas funciones sumara no solo dos parámetros, sino tres o hasta más, habría que dar un nombre diferente a cada una de ellas.

C++ facilita la semántica de este tipo de funciones permitiendo que todas las funciones que queramos puedan llevar el mismo nombre. El único requisito es que

las funciones tengan una *firma* diferente. La firma de la función se conforma con el número, los tipos y el orden de los parámetros (además del nombre de la función, por supuesto). El tipo de dato que devuelve la función no se considera como parte de la firma, por lo que dos funciones que solo se diferencien en este detalle, generarán un error de compilación. El Listado 8 presenta un ejemplo de la sobrecarga de funciones en C++ y, la salida del programa, se muestra a continuación de él:

```
 1.  // Listado 8: sobrecarga.cpp
 2.  // Demuestra la sobrecarga de funciones en C++
 3.  #include <iostream>
 4.  #include <iomanip>
 5.
 6.  using namespace std;
 7.
 8.  // Prototipos de función
 9.  int suma(int a, int b);
10.  int suma(int a, int b, int c);
11.  float suma(float a, float b);
12.  float suma(float a, float b, float c);
13.  double suma(double a, double b);
14.  double suma(double a, double b, double c);
15.
16.
17.  int main(void)
18.  {
19.      int i1 = 10;
20.      int i2 = 20;
21.      int i3 = 30;
22.
23.      cout << "\n- Suma int -\nLa suma de " << i1 << " y "
24.           << i2 << " es: " << suma(i1, i2);
25.
26.      cout << "\nLa suma de " << i1 << ", "
27.           << i2 << " y " << i3 << " es: "
28.           << suma(i1, i2, i3) << endl;
29.
30.      float f1 = 10.0F;
31.      float f2 = 20.0F;
32.      float f3 = 30.0F;
33.
34.      cout << "\n- Suma float -\nLa suma de "
35.           << setprecision(2) << fixed << f1 << " y "
36.           << f2 << " es: " << suma(f1, f2);
37.
38.      cout << "\nLa suma de " << f1 << ", "
39.           << f2 << " y " << f3 << " es: "
40.           << suma(f1, f2, f3) << endl;
41.
42.      double d1 = 10.0;
43.      double d2 = 20.0;
44.      double d3 = 30.0;
45.
```

```
46.      cout << "\n- Suma double -\nLa suma de " << d1 << " y "
47.           << d2 << " es: " << suma(d1, d2);
48.
49.      cout << "\nLa suma de " << d1 << ", "
50.           << d2 << " y " << d3 << " es: "
51.           << suma(d1, d2, d3) << endl;
52.
53.      return 0;
54.
55.  } // main()
56.
57.
58.  // Suma dos números enteros
59.  // a: Primer número a sumar
60.  // b: Segundo número a sumar
61.  int suma(int a, int b)
62.  {
63.      return (a + b);
64.
65.  } // suma()
66.
67.
68.  // Suma tres números enteros
69.  // a: Primer número a sumar
70.  // b: Segundo número a sumar
71.  // c: Tercer número a sumar
72.  int suma(int a, int b, int c)
73.  {
74.      return (a + b + c);
75.
76.  } // suma()
77.
78.
79.  // Suma dos números de tipo float
80.  // a: Primer número a sumar
81.  // b: Segundo número a sumar
82.  float suma(float a, float b)
83.  {
84.      return (a + b);
85.
86.  } // suma()
87.
88.
89.  // Suma tres números de tipo float
90.  // a: Primer número a sumar
91.  // b: Segundo número a sumar
92.  // c: Tercer número a sumar
93.  float suma(float a, float b, float c)
94.  {
95.      return (a + b + c);
96.
97.  } // suma()
98.
99.
100.  // Suma dos números de tipo double
101.  // a: Primer número a sumar
```

```
102.  // b: Segundo número a sumar
103.  double suma(double a, double b)
104.  {
105.      return (a + b);
106.
107.  } // suma()
108.
109.
110.  // Suma tres números de tipo double
111.  // a: Primer número a sumar
112.  // b: Segundo número a sumar
113.  // c: Tercer número a sumar
114.  double suma(double a, double b, double c)
115.  {
116.      return (a + b + c);
117.
118.  } // suma()
```

```
- Suma int -
La suma de 10 y 20 es: 30
La suma de 10, 20 y 30 es: 60

- Suma float -
La suma de 10.00 y 20.00 es: 30.00
La suma de 10.00, 20.00 y 30.00 es: 60.00

- Suma double -
La suma de 10.00 y 20.00 es: 30.00
La suma de 10.00, 20.00 y 30.00 es: 60.00
```

Como podemos observar, la sobrecarga de funciones se utiliza para crear funciones con el mismo nombre que realizan tareas similares, pero sobre tipos de datos diferentes. Lo anterior hace que los programas sean más claros y entendibles.

Con este tema terminamos el repaso al paradigma estructurado en C++, heredado en su mayor parte del lenguaje de programación C. Ahora sí, al mundo de la ***programación orientada a objetos***.

Referencias

[1] Deitel, P. & Deitel, H. (**2014**) C++ How to Program, 9th. Ed. Boston, Massachusetts: Pearson Education, Inc.

[2] S. Horstmann, C. & A. Budd, T. (**2009**) Big C++, 2nd. Ed. New Jersey: John Wiley & Sons, Inc.

[3] Savitch, W. (**2013**) Absolute C++, 5th. Ed. Boston, Massachusetts: Pearson Education, Inc.

[4] E. Walpole, R. & H. Myers, R. & L. Myers, S. & Ye, K. (**2012**) Probabilidad y estadística para ingeniería y ciencias, 9ª Ed. Naucalpan de Juárez, Estado de México: Pearson Educación de México, S. A. de C. V.

[5] Brassard, G. & Bratley, P. (**1997**) Fundamentos de algoritmia. Madrid, España: Pearson Prentice-Hall.

[6] A. Robertson, Lesley (**2004**) Simple Program Design: A Step-by-Step Approach, 4th. Ed. Hong Kong: Course Technology.

[7] S. Malik, D. (**2012**) C++ Programming: Program Design Including Data Structures, 6th. Ed. Boston, Massachusetts: Cengage Learning.

[8] Liberty, J. & Jones, B. (**2005**) Teach Yourself C++ in 21 Days, 5th. Ed. Indianapolis, Indiana, USA: Sams Publishing.

[9] J. Bronson, G. (**2007**) C++ para ingeniería y ciencias, 2ª Ed. México, D. F.: Cengage Learning Editores, S. A. de C. V.

Capítulo 3. Introducción a la *POO* – *Programación Orientada a Objetos*

"… Programar mal es fácil. Los *Tontos* pueden aprender en *21 días*, incluso si son *Dummies*.

Programar bien requiere pensar, pero **todo el mundo** puede hacerlo y **todo el mundo** puede experimentar la satisfacción que viene con ella. Vale la pena pagar el precio por el simple placer del proceso de descubrimiento, la elegancia del resultado y los beneficios comerciales de un proceso sistemático de diseño de programas…"

How To Design Programs, Second Edition
(Cómo diseñar programas, segunda edición)
Matthias Felleisen, Robert Bruce Findler,
Matthew Flatt y Shriram Krishnamurthi

3.1. Introducción

La palabra *paradigma* significa patrón de pensamiento diferente o patrón de conceptualización diferente.[1] Esto es, la forma de visualizar e interpretar los múltiples conceptos, esquemas o modelos.[2] De una manera más reducida, paradigma significa una forma de pensar.

En nuestro caso, el paradigma estructurado definió una forma de pensar con relación a la forma en que el desarrollo de software se debía enfocar, enfrentar, analizar, realizar y evolucionar. Esta forma nueva *de hacer las cosas*, logró impulsar la programación de computadoras en la década de los 70's, más adelante seguirían manteniendo la *fiebre* estructurada el diseño estructurado y el análisis estructurado. Desafortunadamente, con el desarrollo del hardware comandado por la Ley de Moore[3] y el desarrollo de la interfaz gráfica de usuario, hacia la década de los 80's el público demanda aplicaciones de software cada vez más y más complejas. Esto hace que la programación estructurada llegue a su límite y, finalmente, se vea superada.

En esta situación, el paradigma estructurado cedió el paso a un paradigma nuevo que estuviera a la par del gran reto que se presentaba, el paradigma orientado a objetos. Pero como todo cambio, este cambio de paradigma no fue ni ha sido sencillo. La realidad de la *parálisis del paradigma*,[4] es decir, la incapacidad o la negativa a ver más allá del modelo de pensamiento actual (en este caso estructurado), ha hecho que al pensamiento orientado a objetos le tome varias décadas en afianzarse.

Tomemos un ejemplo para *visualizar* la dificultad que presenta un cambio de paradigma. Teniendo las sucesiones numéricas que a continuación se enlistan, determina la metodología o fórmula empleada para acomodar los términos de cada una de ellas:

a) $1, 3, 5, 7, 9, 11, 13, 15, 17, 19$

b) $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \frac{1}{64}, \frac{1}{128}, \frac{1}{256}, \frac{1}{512}$

c) $1, 4, 9, 16, 25, 36, 49, 64, 81, 100$

d) $5, 4, 10, 2, 9, 8, 6, 7, 3, 1$

Seguramente, los tres primeros incisos te habrán resultado fáciles de resolver. Cada uno de ellos implica la multiplicación, suma, división o resta de algún número al anterior en la secuencia, pero no así a la sucesión del cuarto inciso. Esto es porque el paradigma creado por las sucesiones en los tres primeros incisos, no es el adecuado para resolver la cuarta sucesión.

Esta dificultad que se presenta para *romper* nuestro modelo de pensamiento actual, es la misma que se presenta cuando un desarrollador de software, acostumbrado al paradigma estructurado, enfrenta al momento de abordar el paradigma orientado a objetos. Sin embargo, una vez superado este obstáculo, las ventajas de la programación orientada a objetos, sobre la estructurada, comienzan a ser evidentes.

3.2. Ventajas y desventajas del paradigma estructurado y el orientado a objetos

En primer lugar, veamos algunas de las desventajas de la programación estructurada que obligaron a buscarle un reemplazo[5]:

- El enfoque se centra en las funciones (en el *qué hacer*, más que en el *quién lo hace*).
- Tiene costos muy elevados de desarrollo y mantenimiento.
- Tiene una arquitectura rígida, lo que causa que cualquier cambio importante genere un efecto de *onda*, es decir, que afecte a una gran parte del código del programa y que, por lo tanto, se tenga que modificar.
- Tiene muy poca capacidad de generar código reutilizable.

- Se centra en lo técnico (las estructuras de datos y las funciones que las manipulan) y no en el usuario (los objetivos que el usuario tiene al usar el software).

Las ventajas del paradigma orientado a objetos están referidas a subsanar las deficiencias anteriores:[6]

- Reúso: las clases bien diseñadas, de origen, son componentes que se pueden compartir y reutilizar.
- Estabilidad: debido a las características de abstracción y de encapsulación y ocultamiento de información,[i] las clases y los objetos son partes *intercambiables* de software. Solo deben mantener la interfaz pública de servicios.
- Confiabilidad: dado que los componentes individuales de un programa orientado a objetos son menos complejos que los de un programa que implementa exclusivamente el paradigma estructurado, la depuración y el mantenimiento de ellos es más simple, lo que conlleva mayor facilidad para encontrar y eliminar errores y, por consecuencia, mayor confiabilidad integral del software.
- Integridad: la encapsulación y el ocultamiento de información mantienen la protección de los datos que pertenecen a cada objeto, lo cual redunda en la integridad de los mismos.
- Modelado iterativo: el paradigma orientado a objetos permite modelar un número pequeño de clases, luego probar el software, hacer los cambios necesarios, si se requieren, e iterar este procedimiento tantas veces como sea necesario hasta finalizar el desarrollo.

A pesar de estas ventajas, la POO también tiene desventajas:

[i] Los términos nuevos aquí tratados, como abstracción y encapsulación de la información, se abordarán con detalle en apartados posteriores.

- Implica aprender un vocabulario nuevo así como un proceso de pensamiento nuevo.
- La codificación se facilita a cambio de complicar más la etapa de diseño.
- Todavía se puede escribir código fuente complejo y de diseño muy pobre.
- Y, por encima de todo, los requerimientos *aún* cambian constantemente.

Y a todo esto ¿qué es un objeto?

3.3. Objetos y clases

El Diccionario de la lengua española define la palabra objeto como *todo lo que puede ser materia de conocimiento o sensibilidad de parte del sujeto, incluso este mismo*.[7] Los objetos pueden ser muchas cosas; cosas tangibles o concretas como autos, clientes, perros, etc.; cosas intangibles o conceptuales, que son más difíciles de captar o entender, como transacciones, órdenes de compra, demandas judiciales, etc.; y cosas que representan eventos y estados, como fechas, nacimientos, graduaciones, partidas, arribos, aprobaciones, etc.[5] Con relación a la POO, un objeto se define como todo aquello que cuenta con *estado*, *comportamiento* e *identidad*. [8, 9, 10, 11, 12, 13, 14, 15]

Un objeto mantiene información producto de las operaciones que realiza, y esta información influirá en su comportamiento futuro. El conjunto de información que un objeto mantiene se denomina *estado* del objeto.[10] En otras palabras, cada *atributo* de un objeto describe una de las características que lo identifican y tiene un valor único, para un objeto en particular, en un momento en específico. Así pues, el estado de un objeto son los valores de sus atributos en un tiempo dado.[5] Por ejemplo, podemos tener un objeto alumno con los atributos y valores siguientes:

Atributo	Valor
Nombre	José Hernández
Semestre	4
Materias	Programación orientada a objetos, programación cuántica, diseño de portales web, cálculo integral, ética
Calificaciones	10, 9, 8, 6, 9

Entonces, para este momento en específico, el estado de este objeto alumno se conforma de todos los valores en la columna de la derecha de la tabla anterior.

Un objeto es como una variable sofisticada que almacena datos y que puede enviar y recibir mensajes a y desde otros objetos.[ii] En la POO todo es un objeto, incluso un programa es un grupo de objetos enviándose mensajes entre sí, para solicitar u ordenar lo que hay que hacer.

El *comportamiento* de un objeto está definido por las operaciones (llamadas *métodos* en la programación orientada a objetos) que soporta.[10] Esto es, lo que puede hacer el objeto o lo que se le puede hacer al objeto. En este sentido podemos considerar a los objetos como proveedores de servicios. Es decir, un programa provee servicios a los usuarios y, él mismo, emplea servicios proporcionados por otros objetos. Así pues, el objetivo de la POO es producir un conjunto de objetos que oferten los servicios ideales para resolver un problema planteado.[5]

Como ejemplo, para el objeto alumno antes mencionado, su comportamiento podría estar conformado por las operaciones que a continuación se enlistan:

Mostrar nombre	Modificar nombre	Matricular en curso

[ii] Un *mensaje* es una petición para ejecutar un método (función) que pertenece a un objeto en particular.

Mostrar semestre	Modificar semestre	Desmatricular de curso
Mostrar materias	Modificar materias	Pagar inscripción
Mostrar calificaciones	Modificar calificaciones	Aprobar semestre

A pesar de lo anterior, el estado y el comportamiento de un objeto no son suficientes para caracterizar completamente a un objeto. Dos objetos pueden compartir el mismo estado y soportar las mismas operaciones, pero aún así, ser diferentes. Por ejemplo, dos objetos *alumno* del CONALEP (en diferentes planteles de nuestro país) podrían compartir por casualidad el mismo estado, es decir, compartir su nombre, el semestre y las materias que cursan, así como las calificaciones obtenidas en ellas. En tal sentido, el comportamiento para ambos objetos también sería el mismo, dado que ambos objetos son del *tipo alumno*. Por lo tanto, se requiere definir *la identidad* propia de cada objeto, lo cual se puede lograr de varias maneras, por ejemplo, añadiendo un atributo *Matrícula* a ambos objetos.

Y, como se acaba de mencionar, todo objeto debe tener un *tipo*. En la POO, el tipo de un objeto es sinónimo de *clase*. Los objetos que son idénticos, excepto por su estado, se agrupan en *clases de objetos*.[5] En otras palabras, una clase es un conjunto de objetos que comparten una estructura común y un comportamiento común.[8, 9, 10, 11, 12, 13, 14, 15, 16] Por tanto, la definición de una clase debe incluir:

- Las operaciones permitidas sobre los objetos de la clase, y
- Los estados permitidos para los objetos de la clase.

En el diseño de las clases también se debe considerar que todos los aspectos ensamblen apropiadamente. En otras palabras, las clases deben tener un alto grado de *cohesión*. La cohesión significa que la clase tiene la responsabilidad de realizar **una y solo una cosa**, y que tanto sus atributos como sus métodos sustentan esa tarea.[5]

Como consecuencia de todo lo anterior, un objeto es una *instancia* de una clase. Por ejemplo, cuando una alumna nueva, *Sandra*, ingresa a uno de los planteles del CONALEP, se convierte en una instancia de la clase *alumno*.

Cuando se instancia un objeto, se crea un objeto *moldeado* con base en una clase específica. Es decir, la clase es el molde (o plantilla) y el objeto es lo que sale de él. Vale la pena redundar en este enunciado, la clase es la que indica el estado y el comportamiento *válidos* para los objetos que se instanciarán a partir de ella.

3.4. Modelado de objetos

Imagina que estás en tu cuarto de estudio y miras a tu alrededor. Quizás al principio puedas reconocer algunas cosas, como tu cama, tu escritorio o tu televisor, o tal vez tu pecera. También quizás puedas observar tu puerta y tu ventana. Sin embargo, si miras más detenidamente, podrás observar más detalles de cada cosa observada anteriormente, el tamaño de la mesa, el color de la puerta, la posición del escritorio, o la cantidad de suciedad en la pecera. También podrás determinar el propósito de cada uno de estos objetos, como ver la televisión, poner tu computadora y tus útiles escolares sobre el escritorio, o recostarte un momento en tu cama. Y si vamos más allá, quizás te des cuenta de qué material está hecho el escritorio y la potencia de la lámpara, entre otras cosas. Finalmente, quizás también te des cuenta de otros detalles que se pueden percibir con otros sentidos, como el olor de la pecera, la temperatura del cuarto, o el sabor de alguna bebida. También podemos pensar en las cosas que no vemos de estos objetos, como las partes internas de la televisión o de la computadora, la composición química del agua de la pecera, el fabricante de la cama o del escritorio. A pesar de que la información que podemos obtener puede ser bastante, nuestro cerebro cuenta con una propiedad importante conocida como abstracción, que es un proceso mediante el cual podemos reconocer y centrarnos en

las características más importantes de un objeto, para una situación determinada e ignorar aquellas que no son esenciales.[17]

El modelado de objetos, de acuerdo con Grady Booch,[18] tiene varios principios fundamentales, entre ellos están la abstracción y el encapsulamiento.

3.5. Abstracción

Las clases que conformarán la solución del problema que queremos resolver, mediante el desarrollo de software orientado a objetos, deben ajustarse a ciertos principios que permitan desarrollar clases bien diseñadas, que a final de cuentas nos garanticen todos los beneficios de la POO.

El primero de estos principios es la *abstracción*, un proceso mediante el cual se deben definir las características esenciales de un objeto, de tal manera que dichas características puedan diferenciar este de cualquier otro objeto, de manera clara y funcional. Este proceso permite definir las clases sobre las cuales se diseñarán los objetos necesarios para la implementación de un programa orientado a objetos. Mediante este proceso, los objetos del mundo real son representados por objetos abstractos, de los cuales solo se toman los rasgos más importantes, los detalles más significativos y aquellos que son relevantes para la solución del problema y desde el punto de vista del desarrollador de software.

Considera ahora que observas un paisaje. Desde el punto de vista de un artista, este quizás admire los colores y la textura de los árboles. Si fuera un arquitecto, quizás solo observe el mejor lugar para la construcción de alguna vivienda. Si fuera un ecologista, quizás solo le interesen las distintas especies de árboles o plantas o animales que ahí existen. Mientras tanto, si es un niño el que ve el paisaje, quizás solo le interese el árbol en que se podrá subir. Como podemos observar, para cada

uno de estos observadores hay elementos comunes, como los árboles, pero para cada abstracción conceptual hay elementos que solo son importantes desde el punto de vista de cada observador.

Entonces, mediante la abstracción, se aísla la información de los detalles de la implementación y se definen atributos y métodos.[19] Más en concreto, el proceso de abstracción nos permite definir las clases que serán usadas por nuestro programa, de forma análoga a como en la programación estructurada el software está formado por funciones.

La abstracción permite, pues, tener en cuenta únicamente las propiedades pertinentes de un objeto para un problema concreto. Imagina, por ejemplo, que queremos representar a un alumno en sus actividades extraescolares. Entonces el objeto podría tener como atributos: edad, estado de salud, estatura, deporte preferido, etc. Si ahora queremos representar a ese alumno en sus actividades académicas, sus atributos podrían ser: nombre, matrícula, cursos, calificaciones, etc. Como podemos observar, la abstracción que empleamos en la solución, depende del problema que queremos enfrentar.

Con el principio de abstracción la estructura de un problema estará compuesto de objetos comunes[9] que se comunican entre sí a través de mensajes, los cuales darán resultados de acuerdo con la naturaleza de cada objeto.[20]

La abstracción nos permite definir una interfaz para interactuar con los objetos que se encuentran fuera de nuestro sistema. Además, nos permite definir un sistema flexible que tenga la posibilidad de extenderse en formas que, inclusive al momento de diseñarlo, todavía desconozcamos.[5]

3.6. Encapsulación

Otro de los principios que deben regir el diseño de las clases, es la *encapsulación*. El término encapsulación, dentro de la POO, representa aquella propiedad que tienen los objetos de no permitir que sus atributos y métodos sean accesibles de manera externa.[21] Esta propiedad es muy adecuada cuando se realizan programas grandes, ya que se pueden desarrollar módulos de manera independiente y, posteriormente, incluirlos en el proyecto global. Un ejemplo básico de este concepto es el estéreo de un carro, para el conductor el auto le permite seleccionar una estación de radio o reproducir música por algún medio, sin embargo, se le oculta el cómo trabaja el reproductor, pues los componentes del estéreo están ocultos tanto a la vista del conductor como a su manipulación. En este sentido, el encapsulamiento consiste en un proceso mediante el cual se oculta la información y solo se presenta al usuario una interfaz pública (los métodos públicos) para que pueda interactuar con el objeto (ver Figura 1).

Imagina que tienes un objeto A, el cual representa a un estudiante. Supón, además,

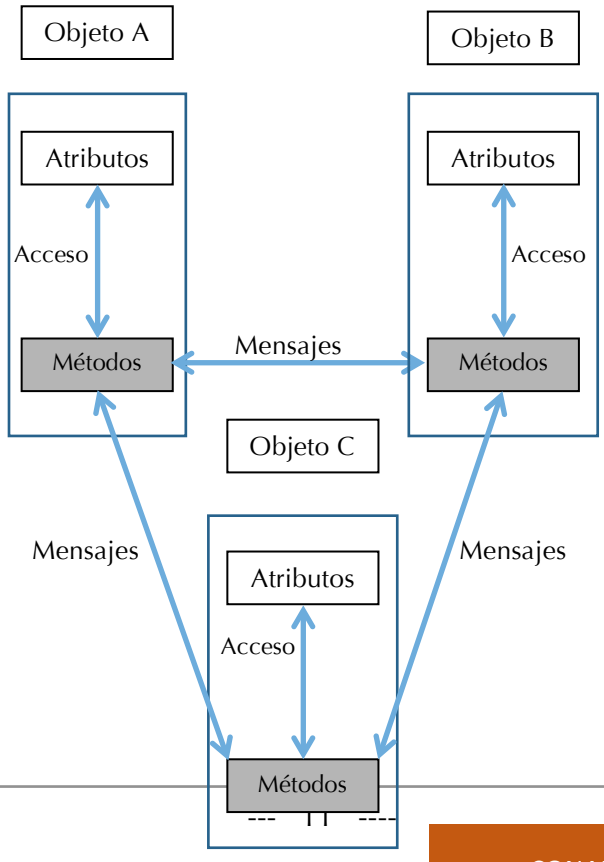

Figura 1. Encapsulación y paso de mensajes

que otro objeto C representa a un curso C. Dado que los objetos se comunican entre sí a través de mensajes, el objeto A le manda un mensaje X al objeto C solicitando los cursos a los que se puede inscribir. Como respuesta, el objeto C le manda un mensaje Y al objeto A indicándole qué tipos de cursos están libres. Tanto al objeto A como al objeto C no les interesa saber cómo está implementado el software que les permite llevar a cabo sus funciones. Ahora bien, si alguien modifica el objeto C para mejorar el tiempo de respuesta, mediante el encapsulamiento el objeto A no sabrá si se ha modificado o no el comportamiento del objeto C, pues solo estará esperando la respuesta de a qué curso se puede inscribir. De esta manera los objetos funcionarán como una caja negra,[22] para los programadores y desarrolladores de sistemas.

El principio de encapsulamiento permite que cada objeto opere como un módulo independiente, haciendo que cada desarrollador trabaje sobre él de manera totalmente aislada y, posteriormente, lo pueda integrar en un solo programa. Otra característica importante del encapsulamiento es ocultar la información y los algoritmos implementados en los métodos. Por lo tanto, el encapsulamiento elimina la necesidad de que otros conozcan el cómo está codificado un objeto, dándole confidencialidad.

Cuando se trabaja con un objeto de una clase en el que tanto sus datos como sus métodos están encapsulados, el usuario del objeto solo dispone de una interfaz pública,[23] mediante la cual puede conocer los servicios que puede solicitar al objeto y saber lo que estos métodos realizan.[iii]

3.6.1. Visibilidad

[iii] Cuando se crea una clase se crea un tipo de datos nuevo, este tipo debe exponer solo lo que es necesario a los clientes a los que les proporcionará algún servicio y mantener oculto todo lo demás. Dicho control de acceso le permite al diseñador cambiar los detalles internos de implementación de la clase, sin preocuparse de que, con esto, pueda afectar a los clientes de la misma (*ocultamiento de información*).[5]

De acuerdo con el principio de encapsulación, se definen entonces dos tipos de control de acceso al contenido de los objetos que son instancias de una clase específica: el privado, destinado para los atributos de la clase, en primer lugar; y, el público, destinado para los métodos de la misma, especialmente los que fungen como parte de la interfaz pública de conexión con el mundo exterior al objeto.

Aunque lo anterior es cierto en primer término, se debe generalizar para poder asignar el tipo de control de acceso o *visibilidad* privado a los métodos que no forman parte de la interfaz pública de la clase, aquellos que solamente asisten en la tarea principal de esta, respetando el principio del privilegio mínimo.[24] De forma análoga se debe poder asignar visibilidad pública a los atributos de la clase, cuando las circunstancias así lo ameriten.[iv] Finalmente, el desarrollo teórico-práctico de la POO ha establecido la utilidad de una tercera forma de visibilidad, tanto para atributos como para métodos, la visibilidad *protegida*.

Entonces, recapitulando, el tipo de control de acceso o *visibilidad* privado implica que solamente los métodos de la misma clase pueden acceder a un atributo o solicitar la ejecución de un método con esta visibilidad. El tipo público significa la disponibilidad, de atributos o métodos, para cualquiera que lo solicite. Por último, la visibilidad protegida implica una visibilidad equivalente a la privada, pero con el agregado de que los objetos que son instancias de clases derivadas de la actual,[v] también tienen acceso directo a los atributos o métodos con este tipo de control de acceso.

[iv] Aunque no se aconseja asignar visibilidad pública a un atributo, existen circunstancias muy particulares en las cuales se podría requerir hacerlo, enfatizando la parte de *muy particulares*.

[v] Las clases derivadas, concernientes a la relación de herencia entre clases en la POO, serán tratadas en apartados posteriores al presente.

3.7. Paso de mensajes

El aspecto básico de la POO es definir clases, de tal manera que los objetos que son instancias de estas clases puedan comunicarse entre sí, a través de la interfaz pública definida por la clase, mediante el envío y recepción de mensajes. Cuando se lleva a cabo el desarrollo de un programa orientado a objetos, este estará conformado de varios objetos entre los cuales el mensaje será el único medio de comunicación, como se ilustra en la Figura 1.

Dicho de otra forma, en la programación estructurada una función recibe los parámetros de entrada para llevar a cabo los procesos, mientras que, en la programación orientada a objetos, el proceso comienza cuando un objeto envía una petición o mensaje a otro objeto en específico. Veamos un ejemplo. En la programación estructurada, si queremos añadir el nombre de un alumno a una lista de datos, necesitamos una función llamada `añadirNombreAlumno` que tome como parámetros de entrada la lista de datos y el propio nombre del alumno. Posteriormente, la función agregará el valor a la lista. Dicha función tendría una forma similar a la siguiente:

```
añadirNombreAlumno(lista, nombreAlumno)
```

Utilizando la programación orientada a objetos, se le envía a un objeto llamado `lista` un mensaje, mediante el cual se le solicita agregue un nombre de alumno a los que ya tiene. Dicho mensaje, en la práctica, lo que solicita es que el objeto `lista` ejecute su método `añadirNombreAlumno`, esto es, se le envía un mensaje al objeto `lista` para que añada un valor. La estructura de la instrucción podría ser similar a la siguiente:[vi]

[vi] La notación *objeto.método*(...) es la forma de expresar en C++ y otros lenguajes orientados a objetos, como Java, que el objeto *objeto* recibió un mensaje, el cual solicita ejecute su método *método*, con los parámetros indicados dentro de los paréntesis.

```
lista.añadirNombreAlumno(nombreAlumno)
```

Es importante mencionar que el comportamiento de los objetos, cuando reciben el mensaje de añadir un nombre de alumno, es particular al objeto que recibe el mensaje. Por ejemplo, si se recibe en un objeto que es una instancia de la clase `Lista`, el nombre del alumno se agregará en orden de lista, si así lo realiza el método ejecutado. Pero, si el objeto receptor es una instancia de la clase `Fila`, el nombre se agregará al final de la lista. Es claro que si se manda el mensaje añadir un nombre de alumno a un objeto que represente una boleta de calificaciones, la función probablemente no tendrá éxito, puesto que a una boleta de calificaciones se le pueden añadir calificaciones, más no nombres de alumnos.

A lo largo de la historia de la POO han surgido diferentes tipos de notación para modelar los diferentes aspectos de los sistemas orientados a objetos que hemos visto hasta el momento (y aún para los que no hemos visto). La notación que a la fecha se ha popularizado y convertido en el estándar de facto para tal objetivo, es el *lenguaje unificado de modelado*.

3.8. El lenguaje unificado de modelado (*UML – Unified Modeling Language*)

El UML es un lenguaje de modelado visual de propósito general que se utiliza para especificar, construir y documentar los *artefactos* de un sistema de software.[25]

El UML proporciona un método y una notación estándares para el modelado de sistemas orientados a objetos, con un enfoque más gráfico que textual, dirigido a la conceptualización y a la abstracción que, además, evoluciona a la par del ciclo de desarrollo de software.

El UML fue desarrollado principalmente por Grady Booch, Ivar Jacobson y James Rumbaugh, como una evolución del trabajo que cada uno había desarrollado individualmente hasta el momento. Al día de hoy, el desarrollo del UML está controlado por el *Grupo de administración de objetos* (*OMG – Object Management Group*), aunque existen variantes y extensiones *no oficiales*.

El UML versión 2.0 fue aprobado por el OMG en 2003 y liberado en 2004. Aunque la mayoría de los cambios se encuentran *tras bambalinas* para el usuario inexperto, esta versión expande la notación y los modelos de la versión anterior, mejora el soporte del uso del UML en la generación automática de código y añade el soporte para la arquitectura manejada por modelos (*model driven architecture*).

El uso principal que se le da al UML es como forma de expresión de los *planos arquitectónicos* de un sistema de software, es decir, los diagramas UML ilustran el diseño del sistema. A pesar de lo anterior, el UML define una gran cantidad de diagramas que sirven para ilustrar vistas (o *modelos*) de la *realidad* diferentes: el modelo de los casos de uso, los modelos estáticos y los modelos de interacción.

El diagrama de casos de uso presenta el modelo inicial del sistema, mostrando una representación gráfica de los servicios que el sistema proporcionará. Se utiliza durante la fase de inicio del proyecto y ayuda a establecer los límites de este. Por el momento no ahondaremos más en este tipo de diagrama.

Los modelos estáticos presentan una *instantánea* del sistema en un punto específico en el tiempo, mostrando la estructura del mismo. Estos modelos son: los diagramas de clases, de objetos, de paquetes, de componentes y de despliegue (deployment).

Los modelos de interacción presentan una visión del sistema en ejecución, mostrando las interacciones que se dan dentro de él y cómo cambia en el tiempo.

Estos modelos son: los diagramas de secuencia, de comunicación (denominados de colaboración en el UML 1.x), de actividad y de máquina de estados.

Prácticamente nos enfocaremos de manera exclusiva en los diagramas de clases, de entre los modelos estáticos del sistema, por ser los más utilizados en general.

Los diagramas de clase muestran las relaciones entre las clases que componen el sistema, siendo el modelo más común. Estos diagramas se pueden mostrar en niveles de abstracción diferentes, esto es, se puede omitir una gran cantidad de detalles en favor de la claridad de lo que se desea transmitir con el diagrama.

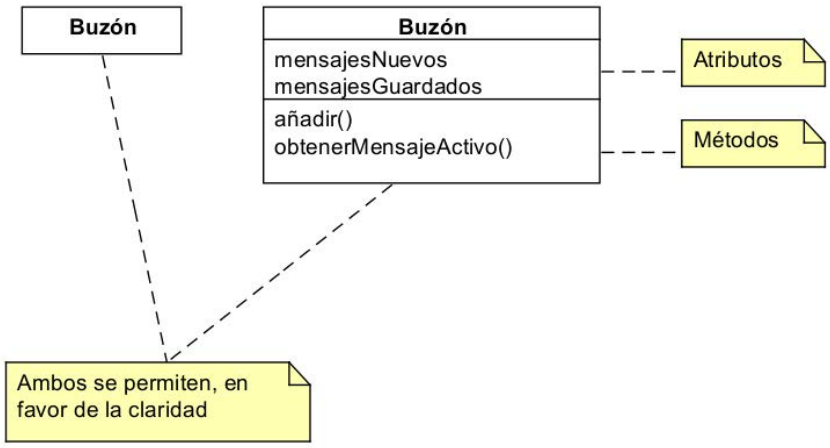

Figura 2. Diagrama de clases en dos niveles diferentes de abstracción

La Figura 2 presenta el diagrama de clases, en dos niveles diferentes de abstracción, de la clase **`Buzón`**. En este diagrama podemos observar que, en UML, una clase se representa como un rectángulo con tres compartimientos: el primero con el nombre de la clase (en negritas), el segundo con los atributos y, el tercero, con los métodos de la clase. Como dijimos anteriormente, los detalles se pueden omitir en favor de la claridad de lo que se quiera expresar con el diagrama. Los rectángulos amarillos con la esquina superior derecha doblada, son la forma de colocar comentarios en el diagrama de clases.

De la misma manera, como lo muestra la Figura 3, podemos modificar el nivel de abstracción para incluir características avanzadas en el diagrama de clases, como etiquetas (metadatos) y anotaciones de visibilidad. Observemos que en UML los atributos y los parámetros de métodos se indican con el patrón `nombre:tipo`. El nombre puede ser cualquiera, equivalente a un identificador de C++, pero el tipo debe corresponder a los tipos especificados por el UML, por ejemplo: Integer (valor entero), Double (valor de punto flotante), Boolean (valor de tipo lógico), String (valor de tipo cadena de caracteres, etc.). De la misma manera, el valor que devuelve cada método se indica con el tipo equivalente del UML; exceptuando el tipo de C++ `void`, que se indica omitiendo por completo los parámetros y/o el tipo del valor de devolución del método.

El concepto de método `constructor` lo trataremos a detalle en una sección subsecuente y, de igual forma, en secciones posteriores presentaremos otros diagramas de clases del UML, cuando tratemos los conceptos relativos a estos.

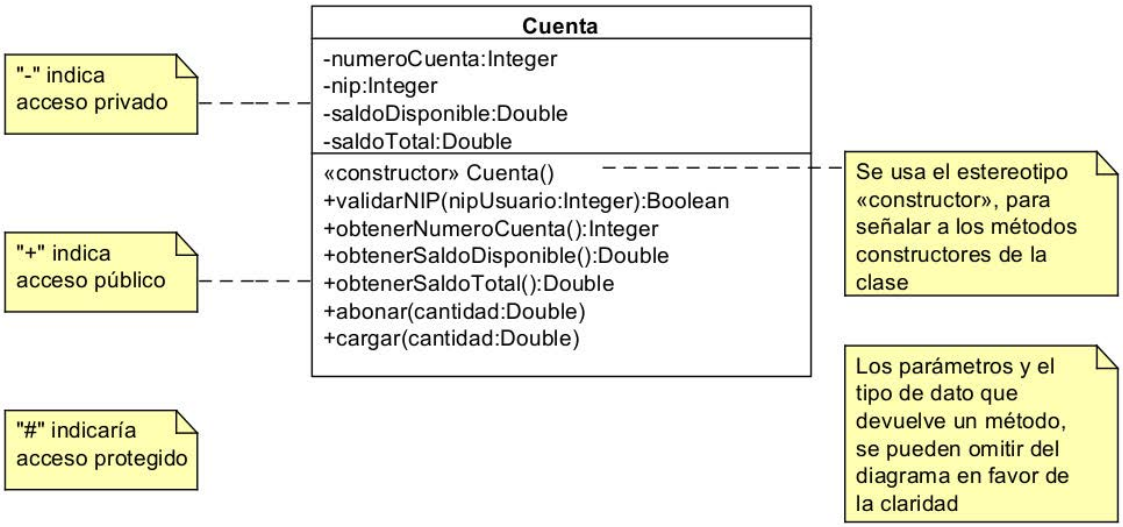

Figura 3. Diagrama de clases con características avanzadas

3.9. Relaciones entre clases

Cuando los objetos interactúan entre sí, generan relaciones entre ellos. Un sistema orientado a objetos está definido por los objetos que lo conforman y las relaciones que existen entre ellos.[6] A continuación veremos las formas principales de relación entre clases y entre objetos que son instancias de esas clases.

3.9.1. Dependencia (*utiliza*)

Una clase *depende* de otra, si la primera manipula objetos de la segunda en alguna forma. Por ejemplo, una clase **`ListaDeAlumnos`** *utiliza* la clase **`Alumno`**, debido a que la clase `ListaDeAlumnos` manipula objetos de la clase `Alumno`.[10]

La dependencia es una relación asimétrica ya que, por ejemplo, la clase `ListaDeAlumnos` utiliza a la clase `Alumno`, pero los objetos de esta última no tienen por qué saber que están agrupados en una lista. Es decir, la clase `Alumno` no depende de la clase `ListaDeAlumnos`. La forma de representar la relación de dependencia en el UML se ilustra en la Figura 4. Si la dependencia es en un solo sentido la línea que conecta ambas clases inicia en la clase dependiente y termina en la otra. Cuando es bidireccional, no se coloca la punta de flecha. En ambos casos es

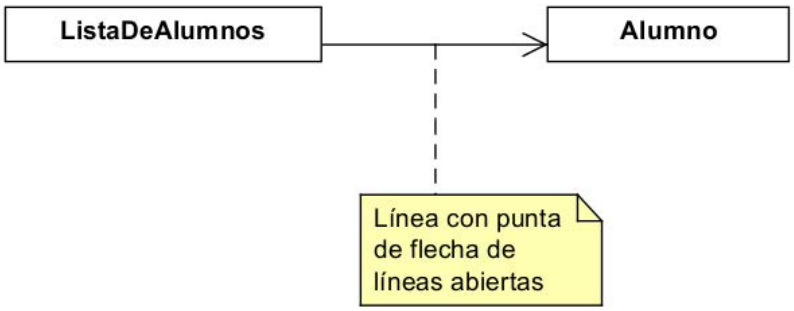

Figura 4. Diagrama de clases de dependencia.

factible etiquetar con *roles* o *papeles* cada extremo de la relación (ver Figura 5).

En el diseño orientado a objetos se debe minimizar el *acoplamiento* entre clases, es decir, las relaciones de dependencia entre ellas. Este acoplamiento se debe minimizar dado que si una clase A depende de una clase B, al modificar la clase B es muy probable que tengamos que modificar también la clase A y, si una tercera clase C dependiera a su vez de A, también debemos modificarla o, al menos, invertir

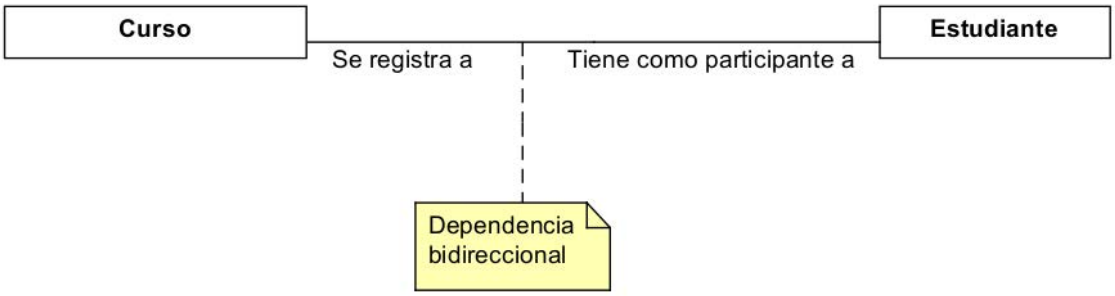

Figura 5. Diagrama de clases de dependencia bidireccional

tiempo valioso en revisar si la clase C aún funciona de la forma que debe, considerando los cambios realizados a la clase A.[5, 10]

En resumen, un grado bajo de acoplamiento entre clases facilita introducir cambios en el sistema en el futuro.

3.9.2. Agregación (*tiene-un*)

El segundo tipo de relación, un caso especial de dependencia, es una forma de reutilización de código (una de las grandes ventajas de la POO), en la cual podemos hacer reúso de la implementación de una clase ya desarrollada y probada. Este tipo de relación se denomina *agregación* y consiste en conformar una clase nueva (la clase que agrega) a partir de clases ya existentes (las clases agregadas). La relación de agregación proporciona flexibilidad al diseño del sistema, ya que los objetos que

conforman a un objeto de la clase recién creada generalmente tienen visibilidad privada, por lo que no son accesibles a los objetos clientes de esta clase. Lo anterior permite, entonces, realizar cambios a las clases agregadas sin necesidad de molestar al código fuente que recibe los servicios de la clase que agrega.[5, 10, 11]

Informalmente, la agregación se describe como la relación *tiene-un*. Una `ListaDeAlumnos` tiene un `Alumno`. En realidad una `ListaDeAlumnos` puede tener muchos objetos de la clase `Alumno`. Al número de objetos de cada clase que intervienen en una relación (por ejemplo, de agregación) se le denomina *multiplicidad*.

Para que un atributo se considere como parte de una relación de agregación, debe ser un objeto instancia de una clase. La mayoría de los lenguajes de programación cuentan con datos de tipo básico o fundamental (por ejemplo los tipos `int`, `float` y `double` de C++), los cuales no generan una relación de agregación entre dos clases.

La Figura 6 muestra la forma de representar en el UML una relación de agregación y las multiplicidades involucradas. El diamante vacío indica cuál es la clase que agrega. En la relación de agregación mostrada, las multiplicidades indican que un objeto de la clase `ListaDeAlumno`, puede tener agregados 0 o más objetos de la clase `Alumno`.

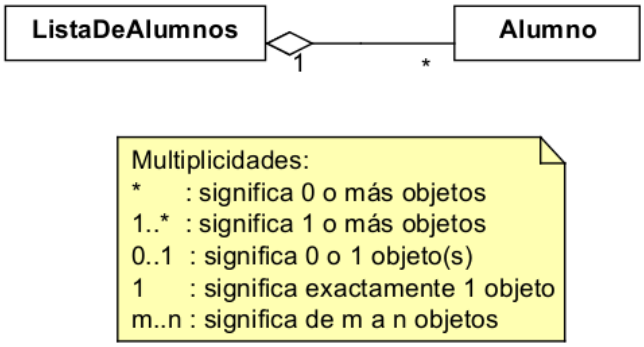

Figura 6. Diagrama de clases de agregación.

3.9.3. Composición (*tiene-un*)

La composición es una forma de agregación, es decir, de relación *tiene-un*. Algunos autores, incluso, consideran que no existe diferencia práctica entre agregación y composición, por lo cual eliminan esta última.[6, 10] Los autores que disienten de lo anterior, señalan que en la agregación los componentes de la relación pueden existir independientemente de dicha asociación. Por ejemplo, un auto tiene un volante, tiene llantas y tiene puertas; sin embargo, el objeto volante, los objetos llantas y los objetos puertas pueden existir de manera independiente, aunque no formen parte de la agregación. Mientras tanto, en la composición, los componentes de la relación no pueden existir independientemente de dicha asociación. Por ejemplo, un cuerpo tiene una cabeza, tiene un tronco y tiene extremidades; sin embargo, el objeto cabeza, el objeto tronco y los objetos extremidades no pueden existir independientemente, sin formar parte de la composición.

La Figura 7 muestra la forma de representar en el UML las relaciones de agregación y de composición. Los diamantes huecos y rellenos indican las clases que agregan y las que están compuestas, respectivamente.

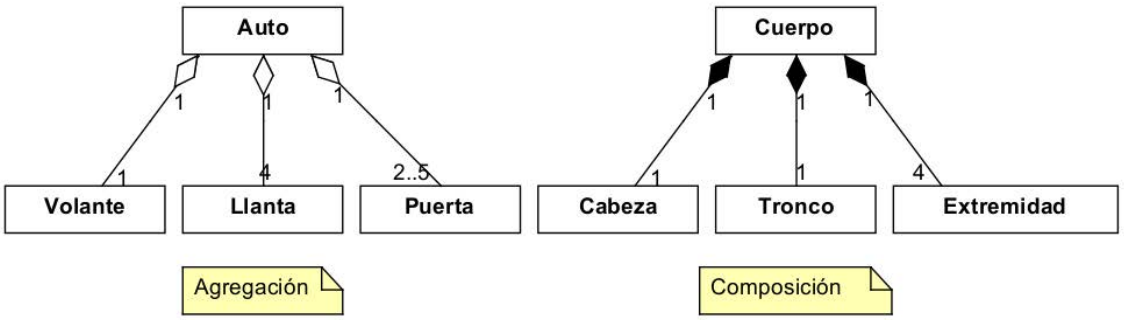

Figura 7. Diagrama de clases de agregación y composición

Las multiplicidades en la agregación indican que un objeto de la clase `Auto`, puede tener agregados un objeto de la clase `Volante`, cuatro objetos de la clase `Llanta`[vii] y de 2 a 5 objetos de la clase `Puerta`. Mientras tanto, las multiplicidades en la composición indican que un objeto de la clase `Cuerpo`, está compuesto de un objeto de la clase `Cabeza`, un objeto de la clase `Tronco` y cuatro objetos de la clase `Extremidad`.

3.9.4. Herencia (*es-un*)

La otra forma de reutilización de código, por medio de relaciones entre clases, es la *herencia*, la cual se define informalmente como la relación *es-un*. Por medio de la herencia podemos *clonar* una clase y modificar el estado o el comportamiento de dicho clon. Si la clase original cambia, la clase *clon* modificada refleja también esos cambios. La clase original se denomina *clase base* (como en C++), *superclase* (como en Java) o, también, *clase padre*. A la clase clon se le suele dar el nombre de *clase derivada* (como en C++), *subclase* (como en Java) o, también, *clase hija*.[5, 6, 10]

[vii] El nombre de cualquier clase debe ser indicado en singular, la multiplicidad indicará el número de objetos involucrados en cada situación.

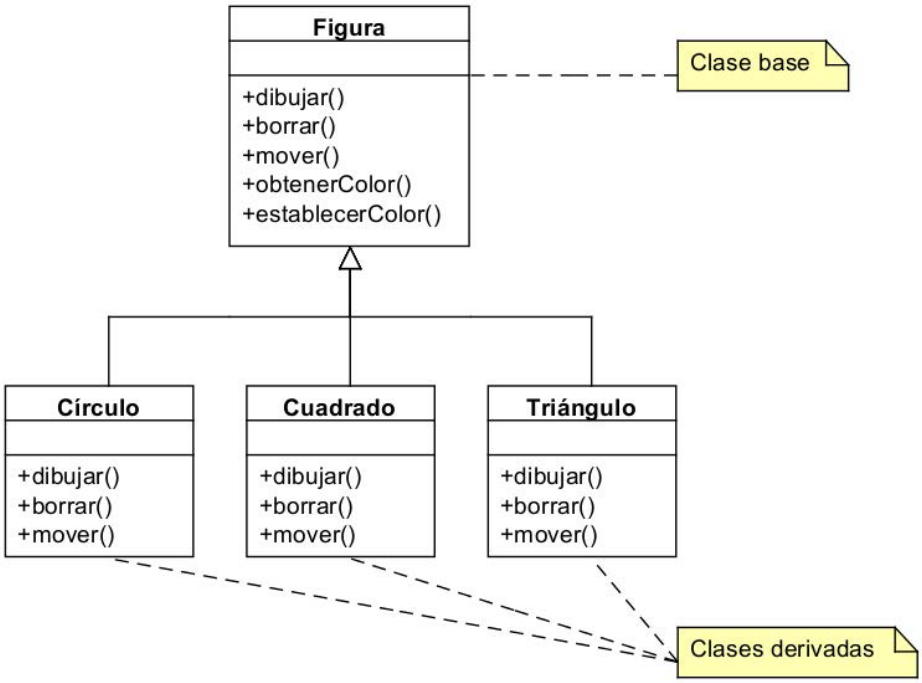

Figura 8. Diagrama de clases de herencia.

Usualmente al emplear herencia tendremos más de una clase derivada. Por ejemplo, la Figura 8 muestra la forma de representar en el UML la relación de herencia y cómo heredan de la clase base `Figura`, las clases `Círculo`, `Cuadrado` y `Triángulo`. Esto es, un objeto de la clase `Círculo` es un objeto de la clase `Figura`; un objeto de la clase `Cuadrado` es una `Figura`; y, un `Triángulo`, es una `Figura`. Debemos observar que la flecha en la relación de herencia apunta hacia la clase más general, es decir, indica el sentido de la relación de generalización.

La clase base contiene todas las características y comportamientos que las clases derivadas comparten. Con la herencia, las clases derivadas definen un nuevo tipo de objetos, los cuales contienen todos los atributos de la clase base más su interfaz pública (todos los métodos públicos de la clase base).

Un detalle en el que cabe la pena redundar, los objetos que son instancias de las clases derivadas también son del tipo de la clase base. Es decir, en la Figura 8, un

objeto `Círculo` también ***es un*** objeto `Figura`; un objeto `Cuadrado` ***es un*** objeto `Figura`; y, un objeto `Triángulo`, ***es un*** objeto `Figura`.

3.9.4.1. Redefinición o sobrescritura de métodos

En el capítulo anterior se presentó la característica denominada sobrecarga de funciones (o métodos), es decir, la creación de funciones/métodos con el mismo nombre que realizan tareas similares, pero sobre tipos de datos diferentes. Además, ya indicamos que una clase derivada puede añadir métodos a los métodos que hereda de una clase base, sin embargo, la POO va un paso más allá, permitiendo que las clases derivadas *redefinan* o *sobrescriban* (*override*) un método heredado de una clase base. [8, 9, 10, 11, 12, 13, 14, 15, 26]

Poniendo un ejemplo de este proceso tenemos que, en la Figura 8, el método `dibujar()` de la clase `Figura` no tiene una razón real de ser, puesto que no podemos dibujar un objeto de una clase tan general como la clase `Figura`, necesariamente debemos saber el tipo concreto de clase cuyo objeto queremos dibujar. Por lo anterior, no es suficiente con que las clases `Círculo`, `Cuadrado` y `Triángulo`, reciban por medio de la herencia, la implementación del método `dibujar()` sino que, más bien, cada una de ellas debe redefinirlo o sobrescribirlo para ajustarlo a sus propias necesidades. Así, cuando se envíe un mensaje "`dibujar`" a un objeto en específico, por ejemplo de la clase `Cuadrado`, este objeto sabrá cómo dibujarse de manera correcta.

3.9.4.2. Clases abstractas y métodos abstractos

Acabamos de mencionar que, en la Figura 8, el método `dibujar()` de la clase `Figura` no tiene razón de ser, por ser `Figura` una clase muy general. Por lo tanto,

su implementación no debe existir en la clase `Figura`. Sin embargo, puesto que la clase base es la que debe contener todo el comportamiento en común que comparten las clases derivadas, este método debe declararse en dicha clase.

La solución que la POO da a la cuestión paradójica anterior, consiste en señalar que el método `dibujar()` de la clase `Figura` es un método *abstracto*. Y, puesto que la clase `Figura` cuenta con, al menos, un método abstracto que carece de implementación, se señala también a esta clase como *abstracta*.

El indicar que la clase `Figura` es una clase abstracta es indicación de su generalidad, es decir, que la definición de sus métodos abstractos no tendría razón de ser, que el requerimiento es que sean sus clases derivadas las que implementen en concreto dichos métodos.

Las clases abstractas así como los métodos abstractos, se representan en el UML como texto en cursivas. La Figura 9 presenta las modificaciones realizadas a la Figura 8, a fin de contemplar que el método `dibujar()` de la clase `Figura` y, junto con

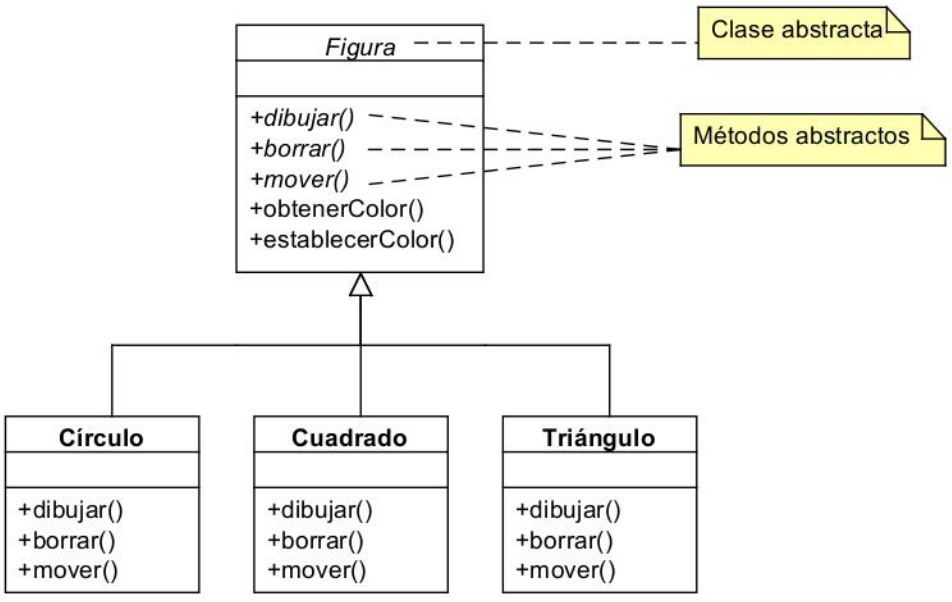

Figura 9. Diagrama de clases de herencia, con una clase y tres métodos abstractos.

él, también los métodos `borrar()` y `mover()` son abstractos.

Resumiendo lo visto en este apartado tenemos que existen dos tipos de herencia: la *herencia completa* y la *herencia de interfase*;[5] y tenemos que existen dos tipos de procedimientos para la reutilización de código por medio de la herencia: la *especialización* y la *generalización*.[27] Además, es importante mencionar que no se pueden instanciar (crear) objetos de una clase abstracta, esto solo se puede realizar de clases *concretas*, es decir, de aquellas clases que cuentan con la implementación de todos los métodos declarados por su clase base (aunque sea abstracta) y por ella misma.

La herencia completa es cuando las clases derivadas heredan la declaración y la definición de los métodos de la clase base. Por ejemplo, en la Figura 9, las clases derivadas reciben la herencia completa de los métodos `obtenerColor()` y `establecerColor()` de la clase `Figura`. La herencia de interfaz es cuando las clases derivadas reciben solamente la declaración de los métodos de la clase base, es decir, que tales métodos son abstractos (y, por lo tanto, también lo es la clase base). En la Figura 9, las clases derivadas reciben la herencia de interfaz de los métodos `dibujar()`, `borrar()` y `mover()`.

La especialización es el proceso de extender una clase existente añadiéndole características por medio de la herencia. La generalización es el proceso de eliminar información y código duplicado en varias clases, mediante la creación de una clase base común a todas ellas (usualmente abstracta), en la cual ubicar tanto la información como el código duplicado.

3.9.4.3. El principio de substitución de Liskov

En los comienzos del auge de la POO, los desarrolladores de software asombrados por la capacidad de reutilización de código que la herencia aportaba empezaron a abusar de ella. Fue así que, en lugar de avanzar la aceptación y práctica de este paradigma nuevo, se detuvo e, incluso, retrocedió. Posteriormente, estudios diversos determinaron situaciones comunes en las que es benéfico hacer uso de la herencia y, también, aquellas otras en las que no.

Uno de los preceptos que mejor resumen las situaciones bajo las cuales es conveniente utilizar la herencia es el *principio de substitución de Liskov* (*LSP*):[28] "Es aceptable derivar la clase B de la clase A solamente si, para cada método en las interfaz públicas de A y B, los métodos de B aceptan como entrada todos los valores que los métodos de A aceptan (y posiblemente más) y, a la par, realizan sobre esos valores todo lo que los métodos de A realizan sobre ellos (y posiblemente más)".

En otras palabras, un objeto de la clase derivada B debe poder substituir a un objeto de la clase base A, en cualquier situación y reaccionando de la misma manera que un objeto de la clase A lo haría. Por ejemplo, ¿es apropiado derivar la clase `Cuadrado` de la clase `Rectángulo`?

Atendiendo a la prueba de herencia *es-un*, resulta que un `Cuadrado` sí es un `Rectángulo` (cuyo alto y ancho miden lo mismo). Desafortunadamente, un rectángulo tiene medidas de alto y ancho, mientras que el cuadrado solo tiene medida de lado, por lo que el LSP dice que no es apropiado derivar la clase `Cuadrado` de la clase `Rectángulo`. Veamos por qué.

La clase `Rectángulo` debe contar con un método que modifique sus medidas, `establecerTamaño(anchoNuevo, altoNuevo)`, garantizando que después de

su ejecución, el rectángulo medirá `anchoNuevo` por `altoNuevo`. A través de la herencia la clase `Cuadrado` recibirá este mismo método pero ¿cómo puede garantizar un objeto de esta clase que después de ejecutar, por ejemplo, `establecerTamaño(10, 15)`, su ancho sea 10 y su alto 15? Por supuesto, ¡no puede! Esta es la razón por la que el LSP no aprueba esta jerarquía de clases.

La nota de advertencia es que, basándonos en el ejemplo anterior, debemos ser cuidadosos al emplear los mecanismos que la POO nos proporciona, especialmente con la herencia de clases. Busquemos que los principios del buen diseño orientado a objetos nos ayuden, como por ejemplo, el LSP.

3.9.5. Polimorfismo

La última relación entre clases que veremos es el polimorfismo. La palabra *polimorfismo* es una palabra de origen griego que significa *muchas formas*. Con relación a la POO, el polimorfismo significa que los objetos que son instancias de clases diferentes, pero todas ellas con una clase base en común, pueden proveer un servicio como respuesta al mismo mensaje, es decir, cada objeto responde a su manera.

Los objetivos del polimorfismo son:

- Minimizar la dependencia entre componentes.
- Que la clase *cliente* (la que requiere) del servicio no necesite saber a qué clase pertenece la clase *servidor* (la que oferta el servicio).
- Que el añadir objetos de una clase nueva que proporcione el servicio no provoque un cambio en las clases clientes, y
- Que el mismo modelo de solución se pueda utilizar en aplicaciones diferentes.

Por ejemplo, dada la especificación en el UML de la clase `Empleado` mostrada en la Figura 10, ¿cómo implementamos el cálculo del salario y los impuestos?

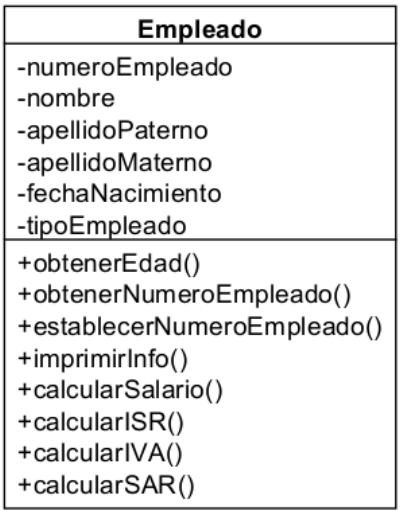

Figura 10. Diagrama UML de la clase Empleado

Si empleamos la programación estructurada, las funciones que realizan estos cálculos lucirían de una forma similar a la siguiente (con la codificación en C++):

```
double calcularSalario()
{
    double salario = 0.0;

    if (tipoEmpleado == GERENTE)
    {
        salario = 50000.0;
    }
    else if (tipoEmpleado == SUBGERENTE)
    {
        salario = 40000.0;
    }
    else if (tipoEmpleado == JEFE_DE_PROYECTO)
    {
        salario = 30000.0;
    }
```

```
    else if (tipoEmpleado == JEFE_DE_DEPARTAMENTO)
    {
        salario = 20000.0;
    }
    else // tipoEmpleado == EMPLEADO_NORMAL
    {
        salario = 10000.0;
    }

    return salario;

} // calcularSalario()


double calcularISR()
{
    double isr = 0.0;

    if (tipoEmpleado == GERENTE)
    {
        isr = 5000.0;
    }
    else if (tipoEmpleado == SUBGERENTE)
    {
        isr = 4000.0;
    }
    else if (tipoEmpleado == JEFE_DE_PROYECTO)
    {
        isr = 3000.0;
    }
    else if (tipoEmpleado == JEFE_DE_DEPARTAMENTO)
    {
        isr = 2000.0;
    }
    else // tipoEmpleado == EMPLEADO_NORMAL
    {
        isr = 1500.0;
    }
```

```
    return isr;

} // calcularISR()


double calcularIVA()
{
    double iva = 0.0;

    if (tipoEmpleado == GERENTE)
    {
        iva = 500.0;
    }
    else if (tipoEmpleado == SUBGERENTE)
    {
        iva = 400.0;
    }
    else if (tipoEmpleado == JEFE_DE_PROYECTO)
    {
        iva = 300.0;
    }
    else if (tipoEmpleado == JEFE_DE_DEPARTAMENTO)
    {
        iva = 200.0;
    }
    else // tipoEmpleado == EMPLEADO_NORMAL
    {
        iva = 175.0;
    }

    return iva;

} // calcularIVA()


double calcularSAR()
{
```

```
        double sar = 0.0;

        if (tipoEmpleado == GERENTE)
        {
            sar = 2500.0;
        }
        else if (tipoEmpleado == SUBGERENTE)
        {
            sar = 2400.0;
        }
        else if (tipoEmpleado == JEFE_DE_PROYECTO)
        {
            sar = 2300.0;
        }
        else if (tipoEmpleado == JEFE_DE_DEPARTAMENTO)
        {
            sar = 2200.0;
        }
        else // tipoEmpleado == EMPLEADO_NORMAL
        {
            sar = 1500.0;
        }

        return sar;

    } // calcularSAR()
```

Ahora bien, ¿qué cambios debemos realizar a los cálculos si se añade el tipo de empleado `EMPLEADO_DE_CONFIANZA`?

La respuesta es que, en cada una de las funciones que realizan los cálculos, debemos agregar una cláusula `if (tipoEmpleado == EMPLEADO_DE_CONFIANZA)` colocando, además, las instrucciones necesarias para asignar a las variables `salario`, `isr`, `iva` y `sar` el monto apropiado para este tipo de empleado.

Como nos podemos imaginar, estas modificaciones son monótonas y muy propensas a errores, existiendo una probabilidad muy alta de que el programador pase por alto actualizar una o varias de estas funciones.

Ahora veamos la forma en que se resolvería el mismo problema, pero empleando la POO y el polimorfismo. Para empezar, debemos crear la implementación de las clases mostradas en la jerarquía de clases de la Figura 11.

Observemos que la clase `Empleado` se ha convertido en abstracta, dado que los métodos `calcularSalario()`, `calcularISR()`, `calcularIVA()` y `calcularSAR()`, de esa clase, se han convertido en abstractos. Además, estos cuatro métodos se redefinen en cada clase derivada. Por lo tanto, la codificación de los cuatro métodos de cada clase quedaría de la forma siguiente (con la codificación en C++):

```
double Gerente::calcularSalario()
{
    return 50000.0;
```

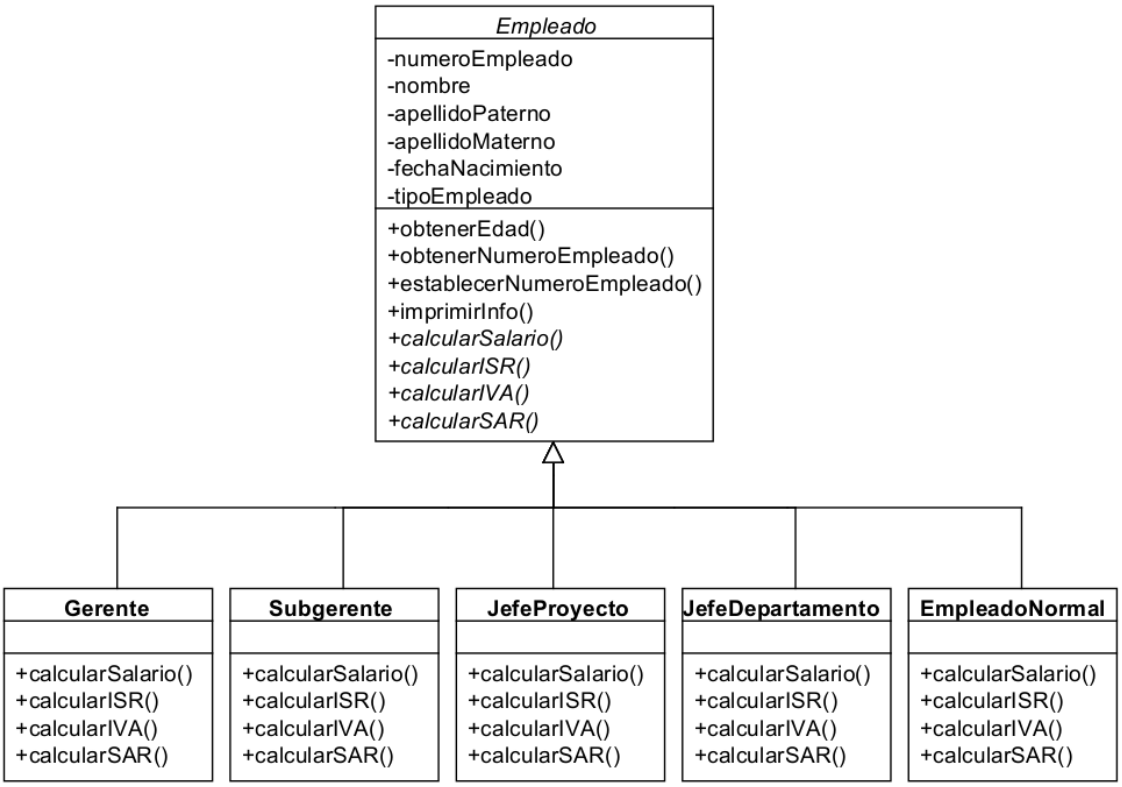

Figura 11. Diagrama UML de la jerarquía de clases de empleados.

```
} // Gerente::calcularSalario()


double Gerente::calcularISR()
{
    return 5000.0;

} // Gerente::calcularISR()


double Gerente::calcularIVA()
{
    return 500.0;

} // Gerente::calcularIVA()
```

```
double Gerente::calcularSAR()
{
    return 2500.0;

} // Gerente::calcularSAR()


double Subgerente::calcularSalario()
{
    return 40000.0;

} // Subgerente::calcularSalario()


double Subgerente::calcularISR()
{
    return 4000.0;

} // Subgerente::calcularISR()


double Subgerente::calcularIVA()
{
    return 400.0;

} // Subgerente::calcularIVA()


double Subgerente::calcularSAR()
{
    return 2400.0;

} // Subgerente::calcularSAR()


double JefeProyecto::calcularSalario()
{
    return 30000.0;
```

```
} // JefeProyecto::calcularSalario()


double JefeProyecto::calcularISR()
{
    return 3000.0;

} // JefeProyecto::calcularISR()


double JefeProyecto::calcularIVA()
{
    return 300.0;

} // JefeProyecto::calcularIVA()


double JefeProyecto::calcularSAR()
{
    return 2300.0;

} // JefeProyecto::calcularSAR()


double JefeDepartamento::calcularSalario()
{
    return 20000.0;

} // JefeDepartamento::calcularSalario()


double JefeDepartamento::calcularISR()
{
    return 2000.0;

} // JefeDepartamento::calcularISR()
```

```
double JefeDepartamento::calcularIVA()
{
    return 200.0;

} // JefeDepartamento::calcularIVA()


double JefeDepartamento::calcularSAR()
{
    return 2200.0;

} // JefeDepartamento::calcularSAR()


double EmpleadoNormal::calcularSalario()
{
    return 10000.0;

} // EmpleadoNormal::calcularSalario()


double EmpleadoNormal::calcularISR()
{
    return 1500.0;

} // EmpleadoNormal::calcularISR()


double EmpleadoNormal::calcularIVA()
{
    return 175.0;

} // EmpleadoNormal::calcularIVA()


double EmpleadoNormal::calcularSAR()
{
```

```
    return 1500.0;

} // EmpleadoNormal::calcularSAR()
```

Visto el código anterior cabe aclarar que, en el lenguaje C++, la notación *NombreClase*`::`*nombreMétodo*`()` indica que el método *nombreMétodo*`()` pertenece a la clase *NombreClase*.

Los cambios en cuanto a código no son muchos, comparándolo contra la versión estructurada, lo importante viene cuando queremos añadir el tipo de empleado `EMPLEADO_DE_CONFIANZA`. En este caso lo único que tendríamos que agregar de código es la definición completa de la clase nueva, derivándola de la clase base `Empleado` y colocando la declaración y definición de los cuatro métodos `calcularSalario()`, `calcularISR()`, `calcularIVA()` y `calcularSAR()`, para esa clase ¡eso es todo! ¡no hay que tocar el código anterior que ya está probado y depurado! El procesamiento polimórfico de la jerarquía de clases se encargará del resto.

Para finalizar este apartado, enlistamos las ventajas de la utilización del polimorfismo de clases:

- Permite escribir código que no depende de tipos específicos.
- El código actual no se afecta por la adición de tipos nuevos (clases nuevas).
- Es una forma esencial para encapsular el código que cambia, y
- Reduce el costo de mantenimiento.

Lo único que debemos considerar para obtener todas las ventajas del polimorfismo, cuando tratamos con jerarquías de tipos, es tratar a los objetos como si fueran del tipo de su clase base, no de su tipo real.

3.10. La cuarta sucesión

Antes de finalizar este capítulo, no queremos dejar completamente en la duda al lector que abordó el ejercicio del apartado inicial, cuyo objetivo era determinar la metodología o fórmula empleada para acomodar los términos de cuatro sucesiones numéricas. Las tres primeras sucesiones son relativamente fáciles de resolver, pero como estas crean en el lector un paradigma que no se aplica a la cuarta sucesión, esta resulta bastante más difícil de resolver. No damos la solución completa, tan solo la pista necesaria para resolverla. La metodología de acomodo de los términos de la sucesión se basa en el *texto* de cada número.

Referencias

[1] Wikipedia. "Paradigm." Consultado: 29 de noviembre de 2013, de http://en.wikipedia.org/wiki/Paradigm

[2] Wikipedia. "Paradigma". Consultado: 29 de noviembre de 2013, de http://es.wikipedia.org/wiki/Paradigma

[3] Intel Corporation. "Moore's Law and Intel Innovation." Consultado: 29 de noviembre de 2013, de http://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html

[4] C. Harrison, J. "Do You Suffer from Paradigm Paralysis?" Consultado: 29 de noviembre de 2013, de http://www.mnsu.edu/comdis/kuster/Infostuttering/Paradigmparalysis.html

[5] Amdocs. "Object Oriented Concepts." Consultado: 29 de noviembre de 2013, de http://pro-cess.co.il/downloads/Dreams_come_true.ppt

[6] Matincor, Inc. "An Introduction to Object-Oriented Analysis and Design using UML." Consultado: 29 de noviembre de 2013, de http://www.matincor.com/Documents/Intro_OOAD.pdf

[7] Real Academia Española. "Diccionario de la lengua española". Consultado: 29 de noviembre de 2013, de http://lema.rae.es/drae/?val=objeto

[8] Booch, G. (**1998**) Análisis y diseño orientado a objetos con aplicaciones, 2ª Ed. Naucalpan de Juárez, Estado de México: Addison Wesley Longman de México, S. A. de C. V.

[9] Deitel, P. & Deitel, H. (**2014**) C++ How to Program, 9th. Ed. Boston, Massachusetts: Pearson Education, Inc.

[10] S. Horstmann, C. (**2006**) Object-Oriented Design & Patterns, 2nd. Ed. Hoboken, New Jersey: John Wiley & Sons, Inc.

[11] S. Horstmann, C. & A. Budd, T. (**2009**) Big C++, 2nd. Ed. New Jersey: John Wiley & Sons, Inc.

[12] Savitch, W. (**2013**) Absolute C++, 5th. Ed. Boston, Massachusetts: Pearson Education, Inc.

[13] S. Malik, D. (**2012**) C++ Programming: Program Design Including Data Structures, 6th. Ed. Boston, Massachusetts: Cengage Learning.

[14] Liberty, J. & Jones, B. (**2005**) Teach Yourself C++ in 21 Days, 5th. Ed. Indianapolis, Indiana, USA: Sams Publishing.

[15] J. Bronson, G. (**2007**) C++ para ingeniería y ciencias, 2ª Ed. México, D. F.: Cengage Learning Editores, S. A. de C. V.

[16] Kimmel, P. (**2007**) Manual de UML. México, D. F.: McGraw-Hill Interamericana Editores, S. A. de C. V.

[17] Barker, J. (**2005**) Beginning Java Objects: From concepts to code, 2nd. Ed. California, USA: Apress Media LLC.

[18] Booch, G. & Cueva L., J. M. & Cernuda del R., A. (**1996**) Análisis y diseño orientado a objetos con aplicaciones. Naucalpan de Juárez, Estado de México: Addison Wesley Longman de México, S. A. de C. V.

[19] Cobo Y., Á. (**2000**) Programar desde un punto de vista científico. Madrid, España: Editorial Visión Libros.

[20] José P., M. (**2004**) C y C++ de afán, 2ª Ed. Antioquía, Colombia: Editorial Universidad de Antioquía.

[21] Martínez G., F. & Martoin Q., G. (**2003**) Introducción a la programación estructurada en C. Valencia, España: Universidad de Valencia.

[22] Ramírez J., F. (**2007**) Aprenda Visual Basic 2005 con Visual Studio 2005. México, D. F.: Pearson Education.

[23] S. Horstmann, C. (**2012**) Big Java, Late Objects. New Jersey: John Wiley & Sons, Inc.

[24] H. Saltzer, J. (1974) Protection and the control of information sharing in multics. Communications of the ACM, Volume 17, Issue 7, July 1974, Pages 388-402. Consulta 29 de noviembre de 2013, de http://dx.doi.org/10.1145/361011.361067

[25] Rumbaugh, J. & Jacobson, I. & Booch, G. (**2005**) The Unified Modeling Language Reference Manual, 2nd. Ed. Boston, Massachusetts: Pearson Education, Inc.

[26] Kimmel, P. (**2007**) Manual de UML. México, D. F.: McGraw-Hill Interamericana Editores, S. A. de C. V.

[27] Skrien, D. (**2009**) Object-Oriented Design Using Java. New York, New York: The McGraw-Hill Companies, Inc.

[28] Liskov, B. "Liskov Substitution Principle." Consultado: 29 de noviembre de 2013, de http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple

Capítulo 4. Implementación de la POO en el lenguaje de programación C++

"… La programación es el arte de expresar soluciones a los problemas de tal manera que una computadora pueda ejecutar esas soluciones. Gran parte del esfuerzo en la programación se dedica a la búsqueda y perfeccionamiento de soluciones. A menudo, un problema se entiende completamente sólo a través del proceso de programar una solución para él…"

Programming: Principles and Practice Using C++
(Programación: Principios y práctica empleando C++)
Bjarne Stroustrup – Creador de C++

En este capítulo veremos las herramientas que proporciona el lenguaje de programación C++ para implementar el paradigma orientado a objetos. Comenzaremos por organizar de una mejor manera el código fuente que escribamos.

4.1. Archivos de cabecera

Cada uno de los ejemplos previos que hemos construido maneja un solo archivo de código fuente con extensión "`.cpp`". Cuando se desarrolla software en un ambiente profesional, es habitual que coloquemos el código fuente reutilizable (como los prototipos de las funciones y, como veremos más adelante, las declaraciones de clases) en un archivo que, por convención, tiene extensión "`.h`". A estos archivos se les denomina *archivos de cabecera* (*headers*) y nos ayudan a organizar el código fuente, de tal manera que es más fácil de entender.

Para ejemplificar este proceso, retomaremos la solución al Problema 2.1, mostrada en el Listado 2 del capítulo anterior, y dividiremos el código en un archivo de cabecera con las declaraciones (Listado 9) y otro con las definiciones (Listado 10), respectivamente. A la par, aprovecharemos para introducir el manejo de memoria dinámica estilo C++.

```
 1.  // Listado 9: estadisticas.h
 2.  // Define los prototipos de función para el archivo estadisticas.cpp
 3.
 4.  // Evita que el archivo de cabecera se incluya más de una vez
 5.  #ifndef ESTADISTICAS_H
 6.  #define ESTADISTICAS_H
 7.
 8.  #include <iostream>
 9.  #include <iomanip>
10.
11.  using namespace std;
12.
13.  // Prototipos de función
14.  int solicitarDatos(int datos[]);
15.  int leerDato(void);
16.  void ordenarDatos(int datos[], int numDatos);
17.  void intercambiar(int& n1, int& n2);
18.  double calcularMedia(int datos[], int numDatos);
19.  double calcularMediana(int datos[], int numDatos);
```

```
20.  int calcularModa(int datos[], int numDatos);
21.  void imprimirDatos(int datos[], int numDatos);
22.
23.  #endif // ESTADISTICAS_H
```

Como podemos observar, el Listado 9 consiste de prácticamente las líneas iniciales del Listado 2, solo se diferencian en el código de las líneas 5, 6 y 23. Estas líneas:

```
#ifndef ESTADISTICAS_H
#define ESTADISTICAS_H
.
.
.
#endif // ESTADISTICAS_H
```

se denominan *guardias de inclusión*, y evitan que el código en el archivo de cabecera se incluya más de una vez en el mismo archivo de código fuente ("`.cpp`"). La primera de estas líneas consulta si el identificador `ESTADISTICAS_H` no está definido, en tal caso, lo define y, además, se incluye todo el código fuente hasta el punto en donde se encuentre la instrucción `#endif`. Si se intentara incluir una segunda vez el mismo archivo de cabecera, ahora sí se encontraría definido el identificador `ESTADISTICAS_H` y el preprocesador (recuérdese que las instrucciones que inician con el símbolo "**#**" corresponden al preprocesador de C++) saltaría todo el código hasta donde encuentre la instrucción `#endif`. También es convención colocar como identificador de los guardias de inclusión el nombre del archivo "`.h`", pero en mayúsculas y con un guion bajo en lugar del punto (ya que dicho carácter no se permite en este contexto).

4.1.1. ¿Cómo se ubican los archivos de cabecera?

```
1.  // Listado 10: estadisticas.cpp
2.  // Calcula la media, la mediana y la moda para un conjunto de datos.
3.  // Imprime el conjunto de datos ordenado y las estadísticas
    calculadas.
4.  #include "estadisticas.h" // Incluir las definiciones personalizadas
5.
```

```
 6.  // Definición de constantes
 7.  const unsigned int MAX_DATOS = 25;
 8.
 9.
10.  int main(void)
11.  {
12.      int *datos = new int[MAX_DATOS]();
13.      int numDatos = solicitarDatos(datos);
14.
15.      if (numDatos > 0)
16.      {
17.          ordenarDatos(datos, numDatos);
18.
19.          double media = calcularMedia(datos, numDatos);
20.          double mediana = calcularMediana(datos, numDatos);
21.          int moda = calcularModa(datos, numDatos);
22.
23.          imprimirDatos(datos, numDatos);
24.          cout << "\nLa media es: " << setprecision(2) << fixed <<
     media
25.               << "\nLa mediana es: " << mediana
26.               << "\nLa moda es: " << moda << endl;
27.      }
28.      else
29.      {
30.          cout << "\nNo se introdujeron datos." << endl;
31.      }
32.
33.      // Devolver la memoria dinámica y asignar un valor
34.      // seguro al apuntador
35.      delete [] datos;
36.      datos = NULL;
37.
38.      return 0;
39.
40.  } // main()
41.
42.
43.  // Solicita y lee datos enteros en el rango [0 - 10]. Un número
     negativo
44.  // finaliza la entrada de datos.
45.  // datos: Arreglo donde se recibirán los elementos solicitados.
46.  // Devuelve el número de datos leído.
47.  int solicitarDatos(int datos[])
48.  {
49.      int numDatos = 0;
50.      int unDato = leerDato();
51.
52.      while (unDato >= 0)
53.      {
54.          datos[numDatos] = unDato;
55.          ++numDatos;
56.
57.          unDato = leerDato();
58.
59.      } // while
```

```
60.
61.      return numDatos;
62.
63.  } // solicitarDatos()
64.
65.
66.  // Lee un dato de tipo entero <= 10. Un número negativo
67.  // finaliza la entrada de datos.
68.  // Devuelve el dato leído.
69.  int leerDato(void)
70.  {
71.      int unDato = 0;
72.
73.      do
74.      {
75.          cout << "\nTeclee un entero x, 0 ≤ x ≤ 10 (-1 para terminar): ";
76.          cin >> unDato;
77.
78.      } while (unDato > 10);
79.
80.      return unDato;
81.
82.  } // leerDato()
83.
84.
85.  // Ordena los 'numDatos' primeros elementos de un arreglo
86.  // datos: Arreglo a ordenar
87.  // numDatos: Número de elementos a ordenar del arreglo
88.  void ordenarDatos(int datos[], int numDatos)
89.  {
90.      for (int i = 0; i < (numDatos - 1); ++i)
91.      {
92.          int minj = i;
93.
94.          for (int j = i + 1; j < numDatos; ++j)
95.          {
96.              if (datos[j] < datos[minj])
97.              {
98.                  minj = j;
99.              }
100.
101.         } // for j
102.
103.         intercambiar(datos[i], datos[minj]);
104.
105.     } // for i
106.
107. } // ordenarDatos()
108.
109.
110. // Intercambia el valor de dos variables
111. // n1: Primera variable a intercambiar
112. // n2: Segunda variable a intercambiar
113. void intercambiar(int& n1, int& n2)
114. {
115.     int temp = n1;
```

```
116.      n1 = n2;
117.      n2 = temp;
118.
119.  } // intercambiar()
120.
121.
122.  // Calcula la media de los primeros 'numDatos' elementos de un
      arreglo
123.  // datos: Arreglo de elementos cuya media se va a calcular
124.  // numDatos: Número de elementos del arreglo cuya media se calculará
125.  double calcularMedia(int datos[], int numDatos)
126.  {
127.      double suma = 0.0;
128.
129.      for (int i = 0; i < numDatos; ++i)
130.      {
131.          suma += datos[i];
132.
133.      } // for i
134.
135.      return (suma / numDatos);
136.
137.  } // calcularMedia()
138.
139.
140.  // Calcula la mediana de los elementos de un arreglo
141.  // datos: Arreglo de elementos cuya mediana se va a calcular
142.  // numDatos: Número de elementos del arreglo cuya mediana se
      calculará
143.  double calcularMediana(int datos[], int numDatos)
144.  {
145.      double mediana = 0.0;
146.      int indiceMediana = numDatos / 2;
147.
148.      if ((numDatos % 2) == 1) // El número de datos es impar
149.      {
150.          mediana = datos[indiceMediana];
151.      }
152.      else // El número de datos es par
153.      {
154.          double suma = static_cast<double>(datos[indiceMediana - 1] +
155.                                            datos[indiceMediana]);
156.          mediana = suma / 2;
157.      }
158.
159.      return mediana;
160.
161.  } // calcularMediana()
162.
163.
164.  // Calcula la moda de los primeros 'numDatos' elementos de un
      arreglo
165.  // datos: Arreglo de elementos cuya moda se va a calcular
166.  // numDatos: Número de elementos del arreglo cuya moda se va a
      calcular
167.  int calcularModa(int datos[], int numDatos)
```

```
168.  {
169.      int frecuencias[11] = {0};
170.
171.      // Contar las ocurrencias de cada valor
172.      for (int i = 0; i < numDatos; ++i)
173.      {
174.          ++frecuencias[ datos[i] ];
175.
176.      } // for i
177.
178.      int iMaximo = 0;
179.
180.      // Buscar la moda
181.      for (int i = iMaximo + 1; i < 11; ++i)
182.      {
183.          if (frecuencias[iMaximo] < frecuencias[i])
184.          {
185.              iMaximo = i;
186.          }
187.
188.      } // for i
189.
190.      return iMaximo;
191.
192.  } // calcularModa()
193.
194.
195.  //  Imprime  en  pantalla  los  primeros  'numDatos'  elementos  de  un
     arreglo
196.  // datos: Arreglo a imprimir
197.  // numDatos: Número de elementos a imprimir del arreglo
198.  void imprimirDatos(int datos[], int numDatos)
199.  {
200.      cout << "\nDatos = [";
201.
202.      for (int i = 0; i < numDatos; ++i)
203.      {
204.          cout << " " << datos[i];
205.
206.      } // for i
207.
208.      cout << " ]" << endl;
209.
210.  } // imprimirDatos()
```

La línea 4 de este Listado 10 incluye el archivo de cabecera que definimos previamente en el Listado 9, con la diferencia de que el nombre se encuentra entrecomillado y no entre paréntesis angulares, como sucede con los archivos de cabecera de la biblioteca estándar. Este detalle le indica al preprocesador de C++

que busque el archivo de cabecera en el mismo directorio en el que se encuentra el archivo "`.cpp`" en el cual encontró la directiva `#include`. Si no encuentra el archivo "`.h`" en dicho directorio, entonces buscará en los mismos directorios donde busca los archivos de cabecera de la biblioteca estándar.[1]

4.2. Asignación y liberación de memoria dinámica estilo C++: los operadores `new` y `delete`

En C++, el manejo de la memoria dinámica[i] se realiza a través de los operadores `new` y `delete`. El operador `new` asigna o reserva la memoria necesaria para almacenar un objeto o un arreglo en tiempo de ejecución, y se accesa a través del apuntador que este operador devuelve. En el capítulo sobre funciones, recordamos que un apuntador es una variable que conserva la dirección en memoria de otra variable, en este caso, la dirección que almacena es la dirección de inicio del bloque de memoria que el operador `new` reservó en el *heap*. Cuando la memoria solicitada ya no se ocupa, se desasigna o devuelve al *heap* por medio del operador `delete`.

Por ejemplo, la línea siguiente:

```
int *entero = new int(17); // Asignación e inicialización simultánea
cout << "El valor del entero es: " << (*entero) << endl;
delete entero;
```

asigna el espacio de memoria suficiente para almacenar una variable de tipo entero, la inicializa con el valor entero `17` y, el apuntador a dicha memoria, se asigna al apuntador `entero`. Se imprime, junto con un mensaje al respecto, el valor del

[i] La asignación y liberación de memoria en el área de *almacenamiento libre* o *heap* (una región de memoria asignada a cada programa por el sistema operativo para este fin en específico), para objetos y para arreglos de elementos de tipo básico o definido por el usuario.

entero a través del apuntador[ii] y, finalmente, cuando ya no se ocupa su valor, la memoria asignada se devuelve al *heap* mediante el operador `delete`.

En la línea 12 del Listado 10:

```
int *datos = new int[MAX_DATOS]();
```

declaramos la variable `datos` como un apuntador a entero, el cual inicializamos con un apuntador al primer elemento de un arreglo asignado dinámicamente de `MAX_DATOS` elementos enteros. Los paréntesis que siguen al valor `new int[MAX_DATOS]` inicializan los elementos del arreglo –los tipos numéricos básicos se inicializan a `0`, el tipo `bool` se inicializa a `false`, los apuntadores a `NULL` (un valor que indica que el apuntador por el momento no es válido) y los objetos de una clase se inicializan con su *método constructor por defecto* (concepto que más adelante explicaremos).

En este momento es importante señalar que la liberación de la memoria asignada dinámicamente por el operador `new`, para un arreglo, se debe liberar con el operador `delete` mediante la notación siguiente:

```
delete [] nombre_apuntador;
```

El olvidar los paréntesis cuadrados o brackets en la instrucción anterior, lleva a errores lógicos en el programa, muy difíciles de detectar. Entonces, la línea 35 del Listado 10 libera apropiadamente el arreglo asignado dinámicamente, además, va un paso más allá, al asegurar que el apuntador `datos` no sea utilizado posteriormente de manera incorrecta, asignándole el valor `NULL` en la línea 36.

[ii] La notación `*nombre_apuntador`, devuelve el contenido de la dirección a la que apunta esta variable, interpretándola como la dirección de inicio de un valor del mismo tipo del que se declaró el apuntador. Esto es, si la variable es un apuntador a un entero, el valor devuelto es un entero o, si la variable es un apuntador a un valor de tipo `double`, el valor devuelto será de ese tipo.

El resto del código en el Listado 10 permanece sin cambios por lo que, la salida en pantalla de una ejecución de prueba del programa del Listado 10, es como podía esperarse:

```
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 3
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 4
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 2
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 3
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 2
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 1
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 1
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 2
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 1
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 1
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 2
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 1
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): 1
Teclee un entero x, 0 <= x <= 10 (-1 para terminar): -1

Datos = [ 1 1 1 1 1 1 2 2 2 2 3 3 4 ]

La media es: 1.85
La mediana es: 2.00
La moda es: 1
```

4.3. Creación y empleo de objetos

Para mostrar la manera en que las clases se definen en C++, desarrollaremos un programa que implemente la clase `Complejo`, cuyo diagrama de clases del UML se muestra en la Figura 1. La clase debe servir para realizar cálculos con números complejos, sabiendo que los números complejos tienen la forma:

$$parteReal + parteImaginaria \times i, donde\ i = \sqrt{-1}$$

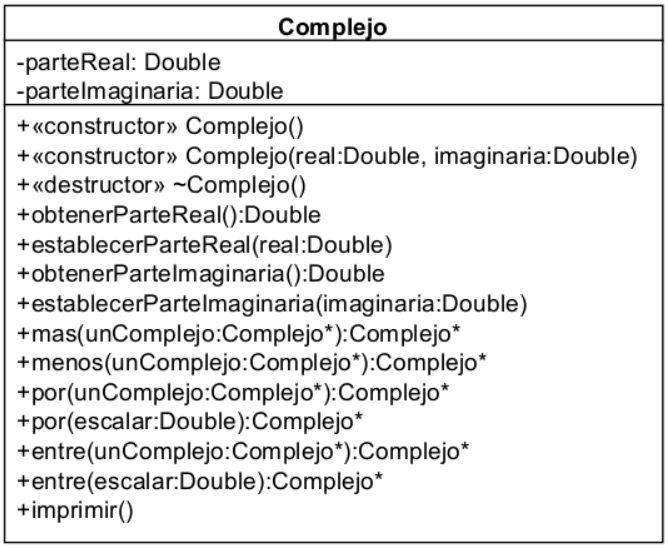

Figura 1. Diagrama de clases del UML de la clase `Complejo`

En esta ocasión mostraremos directamente el programa completo, para dedicarnos a la explicación del mismo y no a su evolución a través del ciclo de desarrollo de software, eso lo dejamos como ejercicio complementario al estudio del lector.

Entonces, tenemos en el Listado 11 la definición de la clase Complejo en C++:

```
1.  // Listado 11: complejo.h
2.  // Definición de la clase Complejo para manejo de números complejos
3.  #ifndef COMPLEJO_H
4.  #define COMPLEJO_H
5.  #include <iostream>
6.
7.  class Complejo
8.  {
```

```
 9.      public:
10.          Complejo();
11.          Complejo(double real, double imaginaria);
12.          ~Complejo() {}
13.          double obtenerParteReal() { return parteReal; }
14.          void establecerParteReal(double real) { parteReal = real; }
15.          double obtenerParteImaginaria() { return parteImaginaria; }
16.          void establecerParteImaginaria(double imaginaria) {
17.              parteImaginaria = imaginaria;
18.          }
19.          Complejo *mas(Complejo *unComplejo);
20.          Complejo *menos(Complejo *unComplejo);
21.          Complejo *por(Complejo *unComplejo);
22.          Complejo *por(double escalar);
23.          Complejo *entre(Complejo *unComplejo);
24.          Complejo *entre(double escalar);
25.          void imprimir(void);
26.      protected:
27.      private:
28.          double parteReal;
29.          double parteImaginaria;
30.
31.  };
32.
33.  #endif // COMPLEJO_H
```

4.3.1. Variables y funciones miembro

Primeramente observamos que este archivo de cabecera también cuenta con los guardias de inclusión vistos anteriormente, en las líneas 3, 4 y 33 del Listado 11. Luego podemos observar que C++ define una clase con la palabra reservada `class` y el nombre de la clase. Es una convención habitual el colocar el nombre de la clase iniciando con una letra mayúscula y el resto del nombre en notación *camello*, es decir, todas las letras en minúsculas excepto la primera letra de cada palabra, cuando el nombre de la clase se conforma de varias palabras, que debe ir en mayúsculas. Ejemplos: `VehículoTodoTerreno`, `ImpresoraLáserColor`, `DepartamentoDeVentas`, etc.

Posteriormente, en las líneas 9, 26 y 27 del Listado 11, podemos observar cómo se especifican los tipos de control de acceso para la encapsulación de los atributos. Por cierto, C++ no emplea la nomenclatura *atributos* y *métodos*, más bien, *variables*

miembro y *funciones miembro*. Entonces, la visibilidad por defecto tanto para variables como para funciones miembro es la privada, es decir, que si no se colocaran los *modificadores de acceso* `public`, `protected` y `private`, todas las características de una clase serían privadas. Por esta razón, lo común es colocar los modificadores en el orden que se ve en el Listado 11.

En la definición de la clase un tipo de control de acceso se mantiene hasta encontrar otro diferente. De acuerdo con esto, la clase `Complejo` no tiene miembros con visibilidad protegida (`protected`); cuenta con solo dos variables miembro que son privadas (`private`) y de tipo `double`: la parte real y la parte imaginaria que conforman al número (objeto) complejo; y todas las funciones miembro son públicas (`public`).

Observemos una cuestión importante en el Listado 11, tenemos funciones miembro que no cuentan con el cuerpo de su definición, es decir, terminan en "**;**". Además, tenemos cinco funciones (contando al destructor, explicado en el apartado siguiente, que tiene un cuerpo vacío) que sí cuentan con su definición completa.

El punto anterior tiene dos motivos de ser: el primero consiste en que solo las funciones miembro que cuentan con una definición muy pequeña se acostumbran colocar en el archivo de cabecera; el segundo motivo se basa en la encapsulación de la información y en el principio del privilegio mínimo, es decir, las clases cliente (y los programadores que las desarrollan) no necesitan ni deberían tener acceso a la implementación de nuestra clase, solamente a la interfaz pública, así, si más adelante podemos optimizar el desempeño de nuestra clase, estaremos en posibilidades de cambiar la forma en que está implementada, sin preocuparnos de afectar a las clases clientes que la utilizan. Por esta razón, la implementación real de las funciones se realiza en un archivo "`.cpp`" aparte, para ocultar la implementación.

4.3.2. Constructores y destructores

Siguiendo con el análisis del Listado 11, las líneas 10 a 25 presentan las funciones miembro de la clase `Complejo`, todas las cuales son de acceso público, como ya dijimos. En el diagrama de clases del UML mostrado en la Figura 1, se utilizó el estereotipo «`constructor`» para resaltar dos funciones miembro (sobrecargadas) con el mismo nombre de la clase y sin valor de devolución de la función. Estas funciones miembros se denominan *constructores*.

Un constructor es una función miembro especial que se llama cuando se instancia (crea) un objeto de la clase, con la finalidad de inicializar las variables miembro y entregar el objeto en un estado apropiado, idealmente antes de su primer uso.

Una clase puede tener más de un constructor, empleando la sobrecarga de funciones, para realizar el proceso de inicialización del objeto según resulte conveniente. Por ejemplo, para la clase `Complejo` uno de los constructores no recibe parámetros y, el otro, recibe como parámetros los valores para inicializar la parte real y la parte imaginaria del número complejo que se está creando. A propósito del constructor que no recibe parámetros, si una clase no incluye *explícitamente* un constructor, C++ le proporciona *implícitamente* un constructor *por defecto*, esto es, un constructor que no recibe parámetros. Este constructor por defecto implícito, no inicializa las variables miembro de tipo básico, pero para aquellas que son objetos de otras clases, llamará a su respectivo constructor por defecto, de tal forma que dichas variables miembro se inicialicen apropiadamente.[1, 2]

Volviendo al diagrama del UML de la Figura 1, la otra función miembro que está marcada con un estereotipo, es el *destructor*. Un destructor es una función miembro especial de una clase que se utiliza para realizar *labores de limpieza* antes de que el

objeto del que forma parte sea destruido. Por ejemplo, si el constructor de la clase asigna espacio dinámicamente para uno o varios objetos que la clase agregue o de la que esté compuesta (o para un arreglo de ellos), el destructor puede emplearse para liberar dicho espacio de memoria, devolviéndolo al sistema operativo. El nombre del destructor para una clase consiste del carácter tilde "**~**" seguido por el nombre de la clase.

Al igual que sucede con los constructores, todas las clases tienen un destructor. Si no proporcionamos el destructor *explícitamente*, C++ proporcionará un destructor *implícito* vacío (sin instrucciones en el cuerpo de su definición).

Para terminar el tema sobre destructores hay que señalar que no se pueden pasar argumentos a un destructor, que no es posible especificarle un tipo de valor de devolución, que no se puede devolver un valor cuando el destructor termina y que tampoco se puede sobrecargar. Si algo de lo anterior se intenta, el resultado será la generación de un mensaje de error por parte del compilador.

4.3.3. Funciones miembro *obtener* (*getters*) y *establecer* (*setters*)

En la Figura 1: *Encapsulación y paso de mensajes* del capítulo anterior, mostrábamos que el acceso a las variables miembro (atributos) de una clase debe realizarse exclusivamente a través de las funciones miembro (métodos) de la interfaz pública proporcionada por la propia clase. Esto debiera ser, incluso, para las propias funciones miembro de la clase. Con esto en mente, la POO designa dos funciones miembro públicas para realizar el acceso a cada variable miembro de una clase: la función *obtener* (en inglés *getter*, o *get* solamente) para leer la variable; y la función *establecer* (*setter*, o *set* solamente) para modificar su valor.

El objetivo es que todo el acceso a las variables internas de la clase se realice a través de las dos funciones mencionadas,[iii] así, cualquier problema que surja con relación a la integridad de las variables será fácil de depurar, ya que el problema se encontrará seguramente en una de dichas funciones. Resulta lógico pensar, por supuesto, que dentro de estas funciones el acceso a las variables miembro es directo.

Cabe señalar que las funciones establecer (*set*) tienen otro propósito muy importante, el de *validar* los valores con los cuales se intentan inicializar las variables miembro. Por ejemplo, si la clase `Persona` maneja la variable miembro `edad`, la función miembro `establecerEdad()` puede y debe validar que esta no sea un número negativo; en tal caso, debe arrojar un mensaje de error y rechazar la modificación, pudiendo asignar un valor predeterminado a la variable citada a fin de dejar en un estado consistente el objeto que se está manipulando.

En las líneas 13 a 16 del Listado 11 vemos las dos funciones obtener, `obtenerParteReal()` y `obtenerParteImaginaria()`, y las dos funciones establecer, `establecerParteReal()` y `establecerParteImaginaria()`, de la clase `Complejo`. Posteriormente, las líneas 19 a 25 muestran la declaración del resto de las funciones miembro de esta clase y, finalmente, las líneas 27 a 29 definen las variables miembro de la clase así como su visibilidad privada.

```
1.  // Listado 12: complejo.cpp
2.  // Definición de las funciones miembro de la clase Complejo
3.  #include <iostream>
4.  #include "complejo.h"
5.
6.  using namespace std;
7.
8.
```

[iii] En ocasiones, por cuestiones de rendimiento, se sugiere realizar el acceso directo a las variables miembro de la clase. También existe quien sugiere que el acceso a estas variables desde los constructores debe realizarse directamente, ya que el objeto en cuestión todavía no cuenta con un estado *inicial*, es decir, el objeto debe construirse completamente antes de utilizarse.

```
 9.  Complejo::Complejo()
10.      : parteReal(0.0), parteImaginaria(0.0)
11.  {
12.  }
13.
14.
15.  Complejo::Complejo(double real, double imaginaria)
16.      : parteReal(real), parteImaginaria(imaginaria)
17.  {
18.  }
19.
20.
21.  Complejo *Complejo::mas(Complejo *unComplejo)
22.  {
23.      return new Complejo(
24.          obtenerParteReal() + unComplejo->obtenerParteReal(),
25.          obtenerParteImaginaria() +
26.              unComplejo->obtenerParteImaginaria()
27.          );
28.
29.  }
30.
31.
32.  Complejo *Complejo::menos(Complejo *unComplejo)
33.  {
34.      return new Complejo(
35.          obtenerParteReal() - unComplejo->obtenerParteReal(),
36.          obtenerParteImaginaria() -
37.              unComplejo->obtenerParteImaginaria()
38.          );
39.
40.  }
41.
42.
43.  Complejo *Complejo::por(Complejo *unComplejo)
44.  {
45.      double a = obtenerParteReal();
46.      double b = obtenerParteImaginaria();
47.      double c = unComplejo->obtenerParteReal();
48.      double d = unComplejo->obtenerParteImaginaria();
49.
50.      return new Complejo(
51.          (a * c) - (b * d),
52.          (a * d) + (b * c)
53.          );
54.
55.  }
56.
57.
58.  Complejo *Complejo::por(double escalar)
59.  {
60.      return new Complejo(
61.          escalar * obtenerParteReal(),
62.          escalar * obtenerParteImaginaria()
63.          );
64.
```

```
65.  }
66.
67.
68.  Complejo *Complejo::entre(Complejo *unComplejo)
69.  {
70.      double a = obtenerParteReal();
71.      double b = obtenerParteImaginaria();
72.      double c = unComplejo->obtenerParteReal();
73.      double d = unComplejo->obtenerParteImaginaria();
74.
75.      return new Complejo(
76.          ((a * c) + (b * d)) / ((c * c) + (d * d)),
77.          ((b * c) - (a * d)) / ((c * c) + (d * d))
78.          );
79.
80.  }
81.
82.
83.  Complejo *Complejo::entre(double escalar)
84.  {
85.      return new Complejo(
86.          obtenerParteReal() / escalar,
87.          obtenerParteImaginaria() / escalar
88.          );
89.
90.  }
91.
92.
93.  void Complejo::imprimir(void)
94.  {
95.      cout << "(" << obtenerParteReal() << ", "
96.           << obtenerParteImaginaria() << ")";
97.
98.  }
```

4.3.4. Implementación de las funciones de una clase

Como ya mencionamos, es costumbre (y necesidad) colocar la implementación de las funciones miembro de una clase en un archivo "`.cpp`" separado del archivo de cabecera. Así pues, el Listado 12 presenta la implementación de estas funciones (al menos, de las que no se implementan en el archivo de cabecera).

En las líneas 3 y 4 del Listado 12 podemos ver la inclusión de los archivos de cabecera de la biblioteca estándar así como de nuestra clase `Complejo`. Luego declaramos el empleo del espacio de nombres estándar, en la línea 6. Más adelante, en las líneas 9, 15, 21, 32, 43, 58, 68, 83 y 93 usamos el operador de resolución de

ámbito "`::`" para señalar que las funciones miembro cuyas cabeceras se encuentran en cada una de esas líneas pertenecen a la clase `Complejo`. Las definiciones de estas funciones miembro abarcan de la línea 9 a la línea 98 del listado en análisis.

4.3.5 Listas de inicialización en los constructores

Las líneas 9 y 10 y 15 y 16 del Listado 12 conforman las cabeceras de las definiciones de los dos constructores sobrecargados de la clase `Complejo`. Ambas cabeceras emplean listas de inicialización para asignar valores directamente a las variables miembro de la clase. La línea:

```
Complejo::Complejo() : parteReal(0.0), parteImaginaria(0.0)
```

indica que la función miembro constructor `Complejo`, pertenece a la clase `Complejo` (empleando el operador de resolución de ámbito "`::`"), que la variable miembro `parteReal` debe inicializarse con el valor `0.0` y que, la variable miembro `parteImaginaria`, debe hacer lo propio también con el valor `0.0`.

Por su parte:

```
Complejo::Complejo(double real, double imaginaria)
    : parteReal(real), parteImaginaria(imaginaria)
```

señala, a su vez, que la función miembro constructor `Complejo`, pertenece a la clase `Complejo` (empleando el operador de resolución de ámbito "`::`"), que dicha función recibe dos parámetros de tipo `double`, el primero de nombre `real` y, el segundo, de nombre `imaginaria`. Además, señala que la variable miembro `parteReal` debe inicializarse con el valor del parámetro `real` y que, la variable

miembro `parteImaginaria`, debe hacer lo propio con el valor del parámetro `imaginaria`.

Observemos que los cuerpos de los constructores están vacíos, ya que los valores de las variables internas ya fueron establecidos con las listas de inicialización.

La función miembro `mas()`, de la clase `Complejo`, se define en las líneas 21 a 29 del Listado 12. Esta función recibe como parámetro un apuntador a un objeto de la misma clase `Complejo`, ya que, siendo `a`, `b` y `c` objetos de esta clase, el uso de la función miembro es:

```
Complejo *a = new Complejo(1, 1);
Complejo *b = new Complejo(2, 2);
Complejo *c = new Complejo();

c = a->mas(b);
```

Recordemos que el operador `new` devuelve un apuntador al área asignada dinámicamente, por lo que debemos usar el operador flecha "**->**" para acceder a las funciones miembro del objeto. Además, con la finalidad de inicializar el objeto recién creado, el operador `new` llamará al constructor correspondiente. Es decir, para las variables `a` y `b` de tipo apuntador a objetos `Complejo`, el constructor invocado será el que recibe dos parámetros y, para la variable `c` de tipo apuntador a objetos `Complejo`, será el constructor por defecto, en otras palabras, el que no recibe parámetros.

Luego entonces, lo que la función `mas()` realiza en las líneas 23 a 27 del Listado 12, es simplemente instanciar un objeto nuevo de tipo `Complejo`, pasándole al constructor que recibe dos parámetros la suma de las partes reales del objeto actual (el que está ejecutando su función `mas()` –el objeto al que apunta la variable `a`, en el ejemplo escrito párrafos arriba) y el objeto al que apunta la variable recibida como

argumento de la función (el objeto al que apunta la variable `b`, en el ejemplo citado). Para terminar con su ejecución, la función `mas()` devuelve el apuntador al objeto construido con la suma (el cual se asignará a la variable `c`, en el ejemplo).

Debemos precisar un detalle del código que implementa la función `mas()` (y de todas las demás funciones miembro de la clase `Complejo`). Cuando queremos accesar uno de los miembros (variable o función) del objeto actual, simplemente colocamos el nombre de la variable (como en las líneas 13, 14, 15 y 17 del archivo de cabecera en el Listado 11) o el de la función correspondiente (como en las líneas 24, 25, 35, 36, 45, 46, 61, 62, 70, 71, 86, 87, 95 y 96 del Listado 12). En cambio, cuando las variables o las funciones miembro que queremos accesar son las del objeto referenciado por una variable apuntador (como el argumento `unComplejo` recibido en las funciones miembro de la clase `Complejo`), debemos usar el nombre de la variable apuntador, el operador flecha y el nombre de la variable o función miembro (como en las líneas 24, 26, 35, 37, 47, 48, 72 y 73 del Listado 12).

Las otras funciones miembro de la clase `Complejo` realizan la misma tarea que la función `mas()` antes descrita, con la salvedad de que aplican una operación diferente (la resta, la multiplicación, la multiplicación por un escalar, la división y la división entre un escalar). La última función miembro, la función `imprimir()`, simplemente presenta el número complejo en pantalla en el formato `(`*`parte_real`*`, `*`parte_imaginaria`*`)`.

Notemos que la implementación de las funciones miembro respeta la encapsulación de los objetos, ya que las variables miembro se accesan a través de los métodos obtener y establecer de la clase, exclusivamente.

Finalmente, como la clase solo define un tipo de datos nuevo (con estado y comportamiento, eso sí), esto es equivalente a saber que existe un tipo de datos

entero `int`. Desafortunadamente, el tipo por sí mismo no nos sirve de mucho, necesitamos desarrollar un programa en el que lo empleemos. De igual manera, para que la definición de la clase `Complejo` nos sea de utilidad, desarrollaremos un programa de prueba pequeño que utilice las facilidades proporcionadas por esta clase. Tal programa se muestra en el Listado 13.

```
 1.  // Listado 13: pruebacomplejo.cpp
 2.  // Programa de prueba de la clase Complejo
 3.  #include <iostream>
 4.  #include "complejo.h"
 5.
 6.  using namespace std;
 7.
 8.  int main(void)
 9.  {
10.      Complejo *a = new Complejo(1, 1);
11.      Complejo *b = new Complejo(2, 2);
12.      Complejo *c = new Complejo();
13.      Complejo *d = new Complejo(3, 5);
14.
15.      cout << "\nSe crean cuatro numeros complejos\na";
16.      a->imprimir();
17.      cout << ", b";
18.      b->imprimir();
19.      cout << ", c y d";
20.      d->imprimir();
21.      cout << endl;
22.
23.      c = a->mas(b);
24.      cout << "\nLa suma de a + b, almacenada en c es:\n\t";
25.      c->imprimir();
26.      cout << endl;
27.
28.      c = d->menos(b);
29.      cout << "\nLa resta de d - b, almacenada en c es:\n\t";
30.      c->imprimir();
31.      cout << endl;
32.
33.      c = b->por(d);
34.      cout << "\nLa multiplicacion de b * d, almacenada en c es:\n\t";
35.      c->imprimir();
36.      cout << endl;
37.
38.      c = a->entre(b);
39.      cout << "\nLa division de a / b, almacenada en c es:\n\t";
40.      c->imprimir();
41.      cout << endl;
42.
43.      c = d->entre(a);
44.      cout << "\nLa division de d / a, almacenada en c es:\n\t";
45.      c->imprimir();
46.      cout << endl;
47.
```

```
48.       return 0;
49.
50.   }
```

En las líneas 10 a 13, del Listado 13, definimos cuatro variables apuntador a objetos de la clase `Complejo`. Tres de ellos los inicializamos al mismo tiempo que el operador `new` asigna memoria dinámica para ellos. Recordemos, otra vez, que el operador `new` llama al constructor de la clase cuyo número de parámetros coincida con el número de parámetros entre los paréntesis de esta instrucción. Posteriormente, en las líneas 15 a 21, imprimimos en pantalla los tres números complejos inicializados.

Más adelante, líneas 23 a 26 del Listado 13, asignamos la suma de dos de los números complejos inicializados al número complejo sin inicializar, y el resultado lo imprimimos en pantalla. Esto mismo lo repetimos en las líneas 28 a 31, para la resta; en las líneas 33 a 36, para la multiplicación; en las líneas 38 a 41, para la división; y, en las líneas 43 a 46, para mostrar otro ejemplo de la división. Finalmente, devolvemos un valor de 0 al sistema operativo, línea 48, indicándole con esto que el programa terminó su ejecución sin contratiempo.

Para complementar el análisis del programa en el Listado 13, a continuación mostramos la salida que presenta en la pantalla:

```
Se crean cuatro numeros complejos
a(1, 1), b(2, 2), c y d(3, 5)

La suma de a + b, almacenada en c es:
        (3, 3)

La resta de d - b, almacenada en c es:
        (1, 3)

La multiplicacion de b * d, almacenada en c es:
        (-4, 16)
```

```
La division de a / b, almacenada en c es:
        (0.5, 0)

La division de d / a, almacenada en c es:
        (4, 1)
```

4.4 Relaciones entre clases

Con la finalidad de mostrar la forma en que el lenguaje de programación C++ implementa las relaciones entre clases, desarrollaremos un programa que aplica el polimorfismo para procesar un arreglo de objetos pertenecientes a diferentes clases en una jerarquía de clases de formas geométricas. La idea es calcular el área de cada uno de los objetos y, cuando la forma geométrica sea tridimensional, calcular también su volumen.

En primer lugar, la Figura 2 presenta el diagrama de clases del UML para la jerarquía de herencia de las formas geométricas.

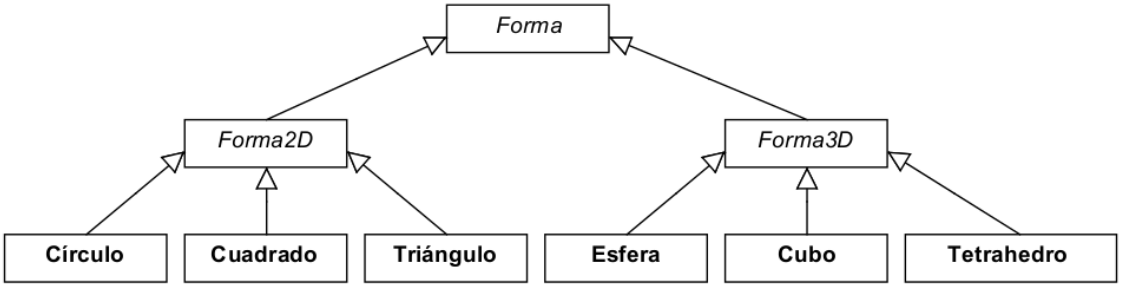

Figura 2. Diagrama de clases del UML para la jerarquía de herencia de las formas geométricas

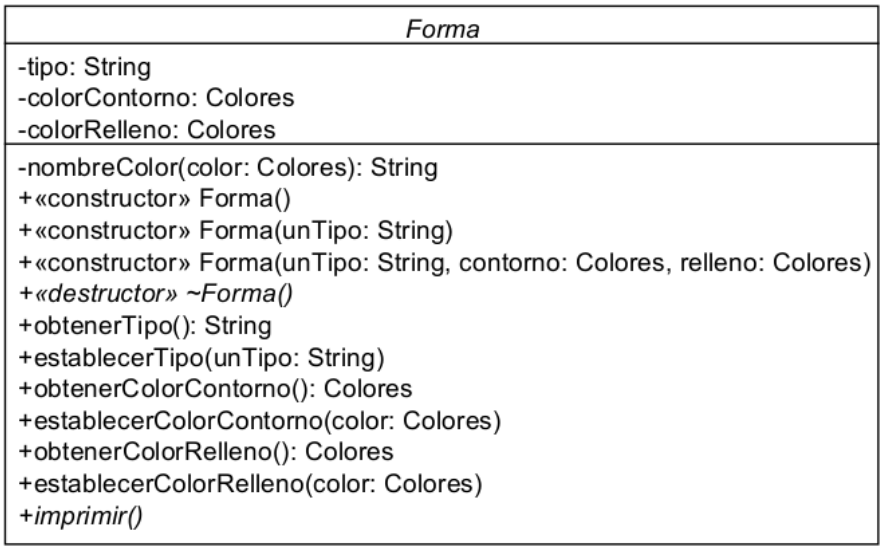

Figura 3. Diagrama de clases del UML de la clase `Forma`

En segundo término, la Figura 3 nos muestra un nivel de abstracción con mayor detalle de la clase `Forma` y, la Figura 4, de las clases `Forma2D` y `Forma3D`.

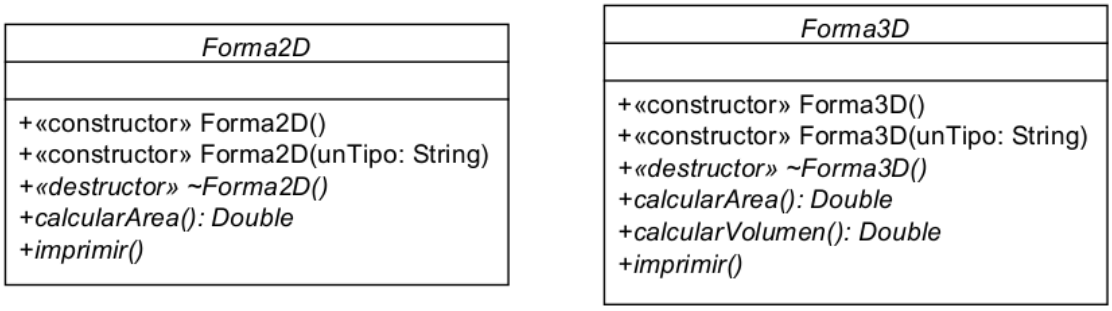

Figura 4. Diagrama de clases del UML de las clases `Forma2D` y `Forma3D`

Observemos que estas tres clases son *abstractas*, como ya se vio en el capítulo anterior, por tener al menos una función miembro abstracta.[iv] En C++, el término para denominar a una función miembro abstracta es *virtual*. Además, cuando la herencia de la función miembro virtual es solamente de interfaz, se denomina *virtual pura*.

[iv] Capítulo 3, sección 3.9.4.2 Clases abstractas y métodos abstractos.

La Figura 5, presenta el mismo diagrama para la clase `Circulo`;[v] la Figura 6, para la

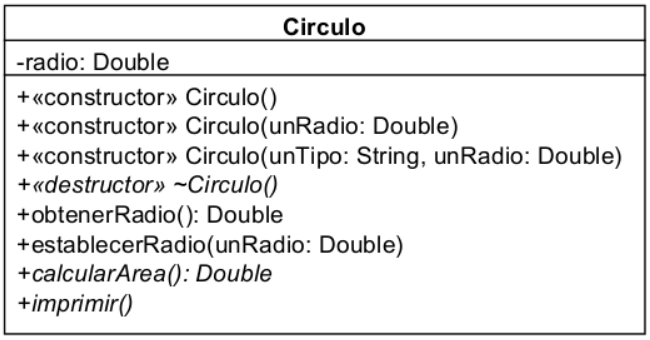

Figura 5. Diagrama de clases del UML de la clase `Circulo`

clase `Cuadrado`; y, la Figura 7, hace lo respectivo para la clase `Triangulo`.

Y, para terminar con la presentación de los diagramas del UML, las Figuras 8, 9 y 10 nos permiten observar a detalle las características de las clases `Esfera`, `Cubo` y `Tetraedro`, respectivamente.

[v] En castellano las palabras *círculo* y *triángulo* se acentúan, pero obtamos por eliminar el acento de ellas en los nombres correspondientes de las clases, a fin evitar problemas sutiles de codificación de caracteres con el compilador de C++.

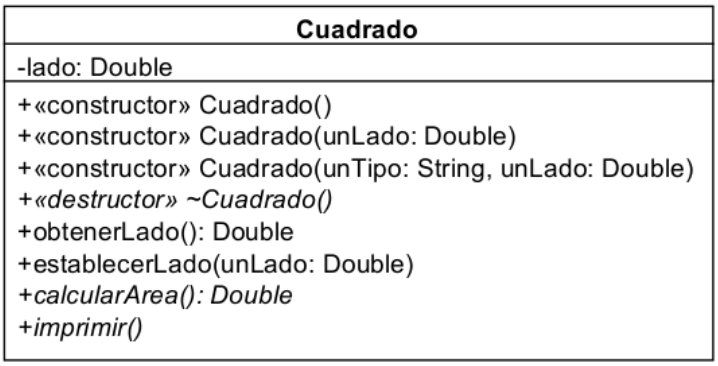

Figura 6. Diagrama de clases del UML de la clase `Cuadrado`

Como en el programa de ejemplo anterior, no detallaremos el ciclo de desarrollo de software completo para este problema, lo dejamos como un *muy* buen ejercicio de

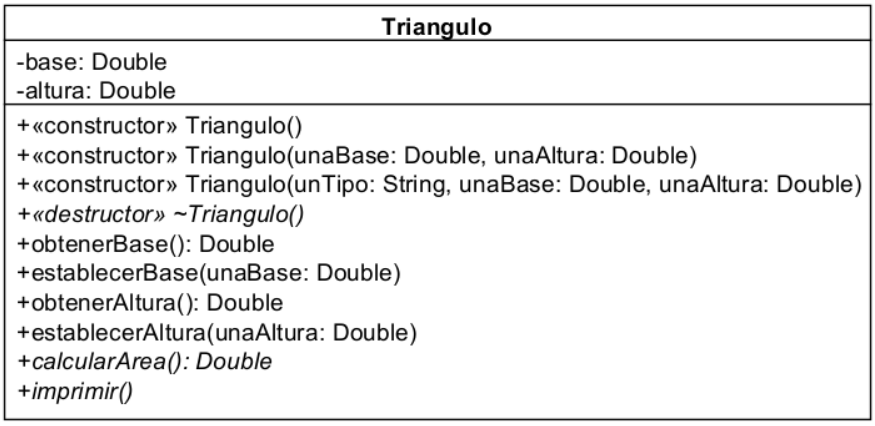

Figura 7. Diagrama de clases del UML de la clase `Triangulo`

práctica para el lector. Entonces, en el Listado 14 se exhibe el primero de los archivos de cabecera del proyecto sobre polimorfismo de las formas geométricas: la definición de la enumeración `Colores`, para el manejo de los colores en los cuales podemos *dibujar* las formas geométricas.

4.4.1. Tipos de datos de enumeración

```
1.  // Listado 14: colores.h
2.  // Definición de la enumeración Colores para manejo de colores
3.  #ifndef COLORES_H
4.  #define COLORES_H
```

```
5.
6.  enum Colores {SIN_COLOR, NEGRO, AZUL, CIAN, GRIS_OBSCURO, GRIS,
7.      VERDE, GRIS_CLARO, MAGENTA, NARANJA, ROSA, ROJO, BLANCO,
8.      AMARILLO};
9.
10. #endif // COLORES_H
```

El Listado 14 declara un tipo de datos definido por el usuario llamado *enumeración*, el cual inicia con la palabra reservada **`enum`**, seguida por un nombre de tipo (en nuestro caso `Colores`) y un conjunto de constantes enteras representadas por identificadores. Los valores de estas constantes inician en 0, si no se indica otro valor para alguna de ellas –como, por ejemplo, `SIN_COLOR = 1`–, y se incrementan de uno en uno. Los nombres de identificadores en la enumeración deben ser únicos, pero el valor de uno o varios de ellos se puede repetir.

A las variables del tipo definido por el usuario `Colores` solo se les puede asignar

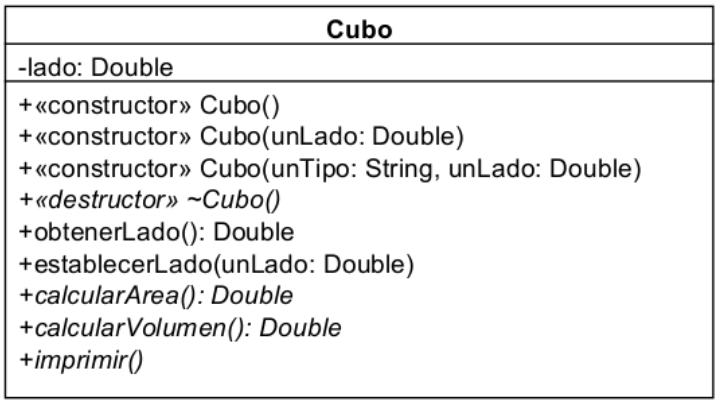

Figura 9. Diagrama de clases del UML de la clase `Cubo`

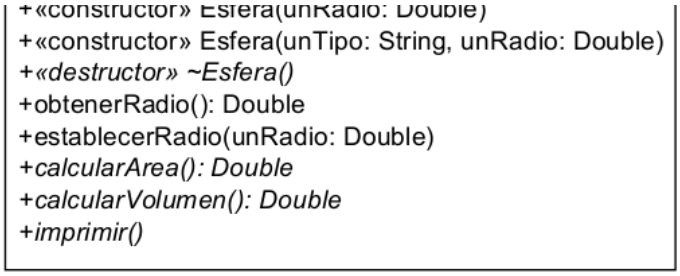

Figura 8. Diagrama de clases del UML de la clase `Esfera`

uno de los valores especificados en la enumeración, utilizando el nombre de la constante, no su valor. Esto es, cualquier variable que declaremos del tipo `Colores`, solo puede recibir un valor como `NARANJA`, `ROSA` o `VERDE`. De igual manera, su valor se puede comparar directamente contra una de estas constantes con nombre. Recordemos que el proporcionar un nombre a una constante se realiza con la finalidad de hacer más explícita su función o su significado. Así pues, emplearemos este tipo de enumeración para asignar un color específico a cada forma geométrica que manejemos, de entre el conjunto de colores que nosotros mismos establecimos (en el tipo enumerado).

Continuando con la implementación de las clases, el Listado 15 muestra el archivo de cabecera de la clase `Forma`.

4.4.2. Constructores `explicit` y destructores `virtual`

```
 1.  // Listado 15: forma.h
 2.  // Definición de la clase Forma
 3.  #ifndef FORMA_H
 4.  #define FORMA_H
 5.
 6.  #include <iostream>
 7.  #include "colores.h"
 8.
 9.  class Forma
10.  {
11.      public:
12.          Forma();
```

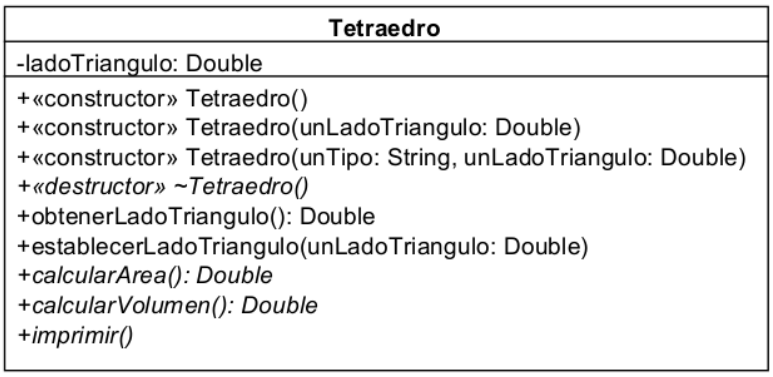

Figura 10. Diagrama de clases del UML de la clase `Tetraedro`

```
13.          explicit Forma(std::string unTipo);
14.          Forma(std::string unTipo, Colores contorno,
15.                Colores relleno);
16.
17.          virtual ~Forma();
18.
19.          std::string obtenerTipo() { return tipo; }
20.          void establecerTipo(std::string unTipo)
21.          { tipo = unTipo; }
22.
23.          Colores obtenerColorContorno() { return colorContorno; }
24.          void establecerColorContorno(Colores color)
25.          { colorContorno = color; }
26.
27.          Colores obtenerColorRelleno() { return colorRelleno; }
28.          void establecerColorRelleno(Colores color)
29.          { colorRelleno = color; }
30.
31.          virtual void imprimir();
32.      protected:
```

```
33.      private:
34.          std::string nombreColor(Colores color);
35.
36.          std::string tipo;
37.          Colores colorContorno;
38.          Colores colorRelleno;
39.  };
40.
41.  #endif // FORMA_H
```

En las líneas 12 a 17 declaramos los constructores y el destructor de la clase `Forma`. Por razones técnicas cuya explicación está más allá del ámbito de este libro, se recomienda que el constructor (que recibe un solo parámetro) de todas las clases que desarrollemos, se declare con la opción `explicit` antes del nombre del constructor. De la misma manera, se recomienda que en toda clase abstracta o que forme parte de una jerarquía de herencia de clases en las que exista alguna clase abstracta, el destructor se declare también como abstracto, es decir, con la palabra reservada `virtual` antes del nombre del constructor.

4.4.3. Más de la clase `Forma`

Las funciones obtener y establecer de la clase `Forma`, se presentan en las líneas 19 a 29 del Listado 15. La línea 31 declara a la función `imprimir()` como abstracta con herencia completa (es decir, se proporciona una implementación de la función, pero también se puede redefinir en las clases derivadas de esta clase). En la línea 34 se define una función miembro con visibilidad `private`, esto es, dicha función ayuda a otras funciones miembro de la clase, pero como no proporciona servicio alguno a otras clases clientes o a las propias funciones miembro de esas clases, no debe dársele acceso `public`, respetando el principio del privilegio mínimo. Finalmente, las líneas 36 a 38 declaran las variables miembro de esta clase.

El Listado 16 nos ofrece la implementación de las funciones miembro de la clase `Forma`.

```
1.  // Listado 16: forma.cpp
2.  // Definición de las funciones miembro de la clase Forma
3.  #include <iostream>
4.  #include "forma.h"
5.  #include "colores.h"
6.
7.  using namespace std;
8.
9.
10. Forma::Forma() : tipo("Forma"), colorContorno(NEGRO),
11.      colorRelleno(SIN_COLOR)
12. {
13.      // Cuerpo vacío
14. }
15.
16.
17. Forma::Forma(string unTipo) : tipo(unTipo), colorContorno(NEGRO),
18.      colorRelleno(SIN_COLOR)
19. {
20.      // Cuerpo vacío
21. }
22.
23.
24. Forma::Forma(string unTipo, Colores contorno, Colores relleno) :
25.      tipo(unTipo), colorContorno(contorno), colorRelleno(relleno)
26. {
27.      // Cuerpo vacío
28. }
29.
30.
31. Forma::~Forma()
32. {
33.      // Cuerpo vacío
34. }
35.
36.
37. // Imprime en la pantalla un objeto de la clase Forma
38. void Forma::imprimir()
39. {
40.      cout << "\nObjeto de tipo: " << tipo
41.           << ".\nColor del contorno: "
42.           << nombreColor( obtenerColorContorno() )
43.           << ".\nColor del relleno: "
44.           << nombreColor( obtenerColorRelleno() ) << "." << endl;
45. }
46.
47.
48. // Transforma un color especificado como constante de enumeración
49. // a su nombre (como texto) correspondiente.
50. // color: La constante de enumeración que especifica el color.
51. // Devuelve el nombre del color.
52. string Forma::nombreColor(Colores color)
53. {
54.      string nombre;
55.
56.      switch(color)
```

```
57.      {
58.      case SIN_COLOR:
59.          nombre = "Ninguno";
60.          break;
61.      case NEGRO:
62.          nombre = "Negro";
63.          break;
64.      case AZUL:
65.          nombre = "Azul";
66.          break;
67.      case CIAN:
68.          nombre = "Cian";
69.          break;
70.      case GRIS_OBSCURO:
71.          nombre = "Gris obscuro";
72.          break;
73.      case GRIS:
74.          nombre = "Gris";
75.          break;
76.      case VERDE:
77.          nombre = "Verde";
78.          break;
79.      case GRIS_CLARO:
80.          nombre = "Gris claro";
81.          break;
82.      case MAGENTA:
83.          nombre = "Magenta";
84.          break;
85.      case NARANJA:
86.          nombre = "Naranja";
87.          break;
88.      case ROSA:
89.          nombre = "Rosa";
90.          break;
91.      case ROJO:
92.          nombre = "Rojo";
93.          break;
94.      case BLANCO:
95.          nombre = "Blanco";
96.          break;
97.      case AMARILLO:
98.          nombre = "Amarillo";
99.          break;
100.     default:
101.         nombre = "Color inexistente";
102.         break;
103.     }
104.
105.     return nombre;
106. }
```

En las líneas 3 a 5, de este listado, se incluyen los archivos de cabecera necesarios para el funcionamiento apropiado de la implementación de las funciones miembro

de la clase `Forma`. Observemos que estamos incluyendo no solo el archivo de cabecera de la clase (`"forma.h"`) sino, también, el que define el tipo de datos enumerado `Colores`, a fin de poder referirnos a los colores del contorno y del relleno de un objeto de la clase `Forma`, por medio de constantes con nombre.

En las líneas 10 a 28 se implementan los tres constructores de esta clase, empleando también listas de inicialización. Proporcionamos tres constructores sobrecargados para dar flexibilidad a nuestro diseño e implementación. Si solo proporcionáramos el constructor que recibe el tipo y los colores del contorno y de relleno del objeto de la clase `Forma`, sería obligatorio colocar estos tres argumentos en cualquier llamada al constructor de esta clase, incluyendo al momento de crear dinámicamente un objeto de esta clase con el operador `new`, aunque este objeto fuera sólo auxiliar en algún cálculo.

En las líneas 31 a 34 presentamos el destructor vacío y, en las líneas 37 a 45, se encuentra la función miembro `imprimir()`, la cual muestra en pantalla un objeto de la clase `Forma`. Observemos que esta es la función que requiere la ayuda de la función miembro privada `nombreColor()`, implementada en las líneas 48 a 106, con la finalidad de convertir el valor enumerado del nombre del color (que en realidad es un número entero) a su representación textual, mediante una estructura de control `switch`.

4.4.4. Herencia

4.4.4.1. La clase `Forma2D`

El Listado 17 presenta el archivo de cabecera de la clase `Forma2D`, la cual se deriva de la clase `Forma`.

```
 1.  // Listado 17: forma2d.h
 2.  // Definición de la clase Forma2D
 3.  #ifndef FORMA2D_H
 4.  #define FORMA2D_H
 5.
 6.  #include <iostream>
 7.  #include "forma.h"
 8.
 9.
10.  class Forma2D : public Forma
11.  {
12.      public:
13.          Forma2D();
14.          explicit Forma2D(std::string unTipo);
15.
16.          virtual ~Forma2D();
17.
18.          virtual double calcularArea() = 0;
19.          virtual void imprimir() override;
20.      protected:
21.      private:
22.  };
23.
24.  #endif // FORMA2D_H
```

En este listado también se manejan los guardias de inclusión, como ya se ha venido manejando. Posteriormente, en las líneas 6 y 7 se incluyen los archivos de cabecera necesarios para las declaraciones que se realizan. Observemos, especialmente, que se incluye el archivo de la definición de la clase `Forma`. Esto es porque en la línea 10 se indica que la clase `Forma2D` se deriva, con herencia de tipo `public`, de la clase ya mencionada `Forma`. La herencia de tipo `private` y `protected` no la estudiaremos en esta obra. Con esta sencilla instrucción, la clase `Forma2D` adquiere todas las variables y funciones miembro de la clase `Forma`. Como podemos observar, éste es uno de los medios de reutilización de código más importantes que existen en la computación.

En las líneas 13, 14 y 16 declaramos los constructores y el destructor de la clase. La línea 18 indica, mediante la palabra reservada `virtual`, que la función miembro `calcularArea()` es abstracta (por lo tanto, también lo es la clase `Forma2D`). Además, señala que la herencia de esta función es solamente de interfaz, como ya se había mencionado anteriormente, con la notación:

```
valor_de_devolución nombre_de_función_abstracta(parámetros) = 0;
```

Lo anterior quiere decir que la clase `Forma2D` no proporciona una implementación de la función `calcularArea()` por lo que, cualquier clase derivada de esta, deberá proporcionar su propia implementación de la función miembro o, de lo contrario, también será una clase abstracta de la cual no es posible instanciar objetos.

En la línea 19, también con la palabra reservada `virtual`, se declara como abstracta la función miembro `imprimir()`. Recordemos del Listado 15 en el cual se define la clase base `Forma`, que esta función `imprimir()` se declara como virtual con herencia de implementación completa, esto es, la clase `Forma` proporciona una implementación de dicha función. Lo que nos lleva a que, en la línea 19 del presente Listado 17, la clase `Forma2D` está redefiniendo la función miembro `imprimir()` de la clase `Forma`, dicho de otro modo, está substituyendo la función de su clase base `Forma`, por su propia función. Esta substitución la hacemos todavía más explícita al colocar la palabra reservada `override`,[vi] al final de la línea 19, la cual es una petición de ayuda al compilador de C++ para indicarle que queremos redefinir la función y que, si hay algo que no nos permita realizar dicho proceso, nos lo indique.

[vi] La palabra reservada `override`, y la funcionalidad que proporciona, solo está disponible en la versión del lenguaje de programación C++ del último estándar C++11. Si el compilador que utilicemos para compilar (valga la redundancia) este código fuente no es compatible con este estándar, el proceso no funcionará.

El Listado 18 nos proporciona el código fuente que implementa las funciones miembro de la clase abstracta `Forma2D`.

```
 1.  // Listado 18: forma2d.cpp
 2.  // Definición de las funciones miembro de la clase Forma2D
 3.  #include <iostream>
 4.  #include "forma2d.h"
 5.
 6.  using namespace std;
 7.
 8.
 9.  Forma2D::Forma2D() : Forma("Forma2D")
10.  {
11.      // Cuerpo vacío
12.  }
13.
14.
15.  Forma2D::Forma2D(std::string unTipo) : Forma(unTipo)
16.  {
17.      // Cuerpo vacío
18.  }
19.
20.
21.  Forma2D::~Forma2D()
22.  {
23.      // Cuerpo vacío
24.  }
25.
26.
27.  // Imprime en la pantalla un objeto de la clase Forma2D
28.  void Forma2D::imprimir()
29.  {
30.      Forma::imprimir();
31.
32.  }
```

En las líneas de la 9 a la 24 de este listado, se incluyen los constructores de la clase `Forma2D`, los cuales utilizan listas de inicialización para realizar todo el proceso de construcción de los objetos de esta clase. En las líneas 21 a 24 se encuentra el destructor vacío.

4.4.4.2. Llamada de funciones miembro de una clase base desde las funciones miembro de una clase derivada

En las líneas 27 a 32 del Listado 18 se encuentra el código de la función miembro `imprimir()`. Como ya analizamos antes, esta función es abstracta y está redefinida en la clase `Forma2D`. Sin embargo, todo el proceso de impresión necesario en esta clase lo realiza, precisamente, la misma función pero de la clase base, luego entonces, la solución es llamar a la versión de la función de la clase base `Forma`. Lo anterior lo conseguimos empleando el operador de resolución de ámbito "`::`" para indicarle a C++ que es a la versión de la clase `Forma` a la que queremos llamar, es decir, la instrucción requerida se muestra en la línea 30 del Listado 18. Esta misma notación la podemos emplear en cualquier situación, y cuantas veces queramos, en que necesitemos llamar a una función de una clase en específico: *`nombre_de_clase`*`::`*`nombre_de_función`*`(`*`parámetros`*`);`

4.4.4.3. Clases concretas derivadas de la clase `Forma2D`

El listado siguiente, el número 19, nos enseña el archivo de cabecera con la definición de la clase `Circulo`, derivada de la clase `Forma2D` y la primera de las clases concretas que implementan todas las funciones virtuales de las clases base `Forma` y `Forma2D`. Las otras son la clase `Cuadrado` y la clase `Triangulo`.

```
 1.  // Listado 19: circulo.h
 2.  // Definición de la clase Circulo
 3.  #ifndef CIRCULO_H
 4.  #define CIRCULO_H
 5.
 6.  #include <iostream>
 7.  #include "forma2d.h"
 8.
 9.
10.  class Circulo : public Forma2D
11.  {
12.      public:
13.          Circulo();
14.          explicit Circulo(double unRadio);
15.          Circulo(std::string unTipo, double unRadio);
16.
```

```
17.          virtual ~Circulo();
18.
19.          double obtenerRadio() { return radio; }
20.          void establecerRadio(double unRadio);
21.
22.          virtual double calcularArea() override;
23.          virtual void imprimir() override;
24.      protected:
25.      private:
26.          double radio;
27.  };
28.
29.  #endif // CIRCULO_H
```

En este listado se incluye en la línea 7 la definición de la clase `Forma2D`, de la cual se deriva la presente clase `Circulo`, situación indicada en la línea 10. Posteriormente, tenemos la declaración de los constructores y el destructor de la clase `Circulo`, en las líneas 13 a 17. Luego tenemos los métodos obtener y establecer en las líneas 19 y 20. Finalmente, tenemos la definición de la función miembro virtual `calcularArea()`, en la línea 22, y la redefinición de la función `imprimir()`, también virtual, en la línea 23. Para terminar, en la línea 26 podemos ver la variable miembro que agrega esta clase.

El listado complementario del anterior, el de la implementación de las funciones miembro de la clase `Circulo`, es el siguiente:

```
 1.  // Listado 20: circulo.cpp
 2.  // Definición de las funciones miembro de la clase Circulo
 3.  #include <iostream>
 4.  #include <cmath>
 5.  #include "circulo.h"
 6.
 7.  using namespace std;
 8.
 9.  const double PI = 3.141592653589793;
10.
11.
12.  Circulo::Circulo() : Forma2D("Circulo"), radio(0.0)
13.  {
14.      // Cuerpo vacío
15.  }
16.
17.
18.  Circulo::Circulo(double unRadio) : Forma2D("Circulo")
19.  {
20.      establecerRadio(unRadio);
```

```
21.  }
22.
23.
24.  Circulo::Circulo(string unTipo, double unRadio) : Forma2D(unTipo)
25.  {
26.      establecerRadio(unRadio);
27.  }
28.
29.
30.  Circulo::~Circulo()
31.  {
32.      // Cuerpo vacío
33.  }
34.
35.
36.  void Circulo::establecerRadio(double unRadio)
37.  {
38.      // Validar el radio
39.      if (unRadio >= 0.0)
40.      {
41.          radio = unRadio;
42.      }
43.      else
44.      {
45.          radio = 0.0;
46.
47.      }
48.
49.  }
50.
51.
52.  // Calcula el área del círculo
53.  double Circulo::calcularArea()
54.  {
55.      return PI * obtenerRadio() * obtenerRadio();
56.
57.  }
58.
59.
60.  // Imprime un objeto de la clase Circulo
61.  void Circulo::imprimir()
62.  {
63.      Forma2D::imprimir();
64.
65.      cout << "Radio: " << obtenerRadio() << ", Area: "
66.           << calcularArea() << endl;
67.
68.  }
```

En este listado definimos la constante matemática π, en la línea 9, requerida para el cálculo del área del círculo. En las líneas de la 12 a la 33, se ubican los constructores y el destructor (vacío) de la clase.

4.4.4.4. Listas de inicialización en constructores de clases derivadas

Podemos ver que en la línea 12 de este Listado 20, el constructor utiliza una lista de inicialización para construir un objeto `Circulo` pero, además, llama al constructor de la clase base, `Forma2D`, para que inicialice la parte del objeto que se está construyendo y que corresponde a la clase `Forma2D`. Es importante que esta llamada al constructor de la clase base se coloque en la primera posición de la lista de inicialización, el no hacerlo así es un error de sintaxis.

```
clase_derivada::constructor_clase_derivada() :
    constructor_clase_base(parámetros), variable_miembro1(valor1),…
    variable_miembroN(valorN){ … }
```

4.4.4.5. Orden de llamada de constructores y destructores

En el apartado anterior vimos que un constructor de una clase derivada puede llamar a uno de los constructores de la clase base para inicializar la parte correspondiente a esa clase, del objeto que se está construyendo. Esta clase, a su vez, puede llamar a un constructor de su propia clase base (si la tuviera, claro está), y así sucesivamente. Esto implica que la construcción de un objeto, en una jerarquía de herencia de clases, se construye de la parte de la clase más general que lo conforma, a la parte de la clase más particular de la cual está constituido.

Para complementar el proceso de construcción descrito, el destructor de cada clase se ejecuta en el orden inverso a como los constructores se ejecutaron, o sea, se *destruye* primero la parte del objeto que corresponde a la clase más particular, y así

sucesivamente, hasta llegar a destruir la parte del objeto que corresponde a la clase más general.

4.4.4.6. Más de la clase `Circulo`

En las líneas 36 a 49 del Listado 20 se encuentra la función miembro `establecerRadio()`, la cual realiza uno de los propósitos principales para el que se emplean este tipo de funciones, es decir, validar los valores que se pretende asignar a las variables miembro. En este caso, la función valida que no se asigne un valor negativo a la variable miembro `radio`. Por este mismo motivo, los constructores de la clase, en las líneas 20 y 26, llaman a la función `establecerRadio()` para asignar el valor que se pretende, y no inicializan la variable miembro directamente desde la lista de inicialización.

Finalmente, en las líneas 52 a 68 del Listado 20, se encuentra la implementación de las funciones miembro `calcularArea()` e `imprimir()`. La primera calcula el área del objeto círculo empleando la fórmula matemática correspondiente y, la segunda, imprime dicho objeto en la pantalla. Observemos cómo se auxilia esta función, en la línea 63, de la función `imprimir()` de su clase base `Forma2D` para realizar el trabajo de impresión. La función `imprimir()` de la clase `Circulo`, solamente añade la impresión de las características que esta clase agrega a la jerarquía de herencia de clases: el radio y el resultado del cálculo del área.

Con la finalidad de evitar el ser repetitivos, a continuación solo presentaremos los listados con el contenido de los archivos de cabecera y de la implementación de las funciones miembro de las clases `Cuadrado` y `Triangulo`. El análisis de estos cuatro listados es similar al de los listados correspondientes a la clase `Circulo` solo varían, por supuesto, en la forma geométrica de que se trata, no en los aspectos de programación que se están presentando.

Primeramente el Listado 21:

```
 1.  // Listado 21: cuadrado.h
 2.  // Definición de la clase Cuadrado
 3.  #ifndef CUADRADO_H
 4.  #define CUADRADO_H
 5.
 6.  #include <iostream>
 7.  #include "forma2d.h"
 8.
 9.
10.  class Cuadrado : public Forma2D
11.  {
12.      public:
13.          Cuadrado();
14.          explicit Cuadrado(double unLado);
15.          Cuadrado(std::string unTipo, double unLado);
16.
17.          virtual ~Cuadrado();
18.
19.          double obtenerLado() { return lado; }
20.          void establecerLado(double unLado);
21.
22.          virtual double calcularArea() override;
23.          virtual void imprimir() override;
24.      protected:
25.      private:
26.          double lado;
27.  };
28.
29.  #endif // CUADRADO_H
```

Ahora el Listado 22:

```
 1.  // Listado 22: cuadrado.cpp
 2.  // Definición de las funciones miembro de la clase Cuadrado
 3.  #include <iostream>
 4.  #include "cuadrado.h"
 5.
 6.  using namespace std;
 7.
 8.
 9.  Cuadrado::Cuadrado() : Forma2D("Cuadrado"), lado(0.0)
10.  {
11.      // Cuerpo vacío
12.  }
13.
14.
15.  Cuadrado::Cuadrado(double unLado) : Forma2D("Cuadrado")
16.  {
17.      establecerLado(unLado);
18.
19.  }
```

```
20.
21.
22.  Cuadrado::Cuadrado(string unTipo, double unLado) :
23.      Forma2D(unTipo)
24.  {
25.      establecerLado(unLado);
26.
27.  }
28.
29.
30.  Cuadrado::~Cuadrado()
31.  {
32.      // Cuerpo vacío
33.
34.  }
35.
36.
37.  void Cuadrado::establecerLado(double unLado)
38.  {
39.      // Validar el lado
40.      if (unLado >= 0.0)
41.      {
42.          lado = unLado;
43.      }
44.      else
45.      {
46.          lado = 0.0;
47.
48.      }
49.
50.  }
51.
52.
53.  // Calcula el área del cuadrado
54.  double Cuadrado::calcularArea()
55.  {
56.      return obtenerLado() * obtenerLado();
57.
58.  }
59.
60.
61.  // Imprime un objeto de la clase Cuadrado
62.  void Cuadrado::imprimir()
63.  {
64.      Forma2D::imprimir();
65.
66.      cout << "Lado: " << obtenerLado() << ", Area: "
67.           << calcularArea() << endl;
68.
69.  }
```

Luego el Listado 23:

```
1.  // Listado 23: triangulo.h
2.  // Definición de la clase Triangulo
```

```
 3.  #ifndef TRIANGULO_H
 4.  #define TRIANGULO_H
 5.
 6.  #include <iostream>
 7.  #include "forma2d.h"
 8.
 9.
10.  class Triangulo : public Forma2D
11.  {
12.      public:
13.          Triangulo();
14.          Triangulo(double unaBase, double unaAltura);
15.          Triangulo(std::string unTipo, double unaBase,
16.                    double unaAltura);
17.
18.          virtual ~Triangulo();
19.
20.          double obtenerBase() { return base; }
21.          void establecerBase(double unaBase);
22.
23.          double obtenerAltura() { return altura; }
24.          void establecerAltura(double unaAltura);
25.
26.          virtual double calcularArea() override;
27.          virtual void imprimir() override;
28.      protected:
29.      private:
30.          double base;
31.          double altura;
32.  };
33.
34.  #endif // TRIANGULO_H
```

Y, finalmente, el Listado 24:

```
 1.  // Listado 24: triangulo.cpp
 2.  // Definición de las funciones miembro de la clase Triangulo
 3.  #include <iostream>
 4.  #include "triangulo.h"
 5.
 6.  using namespace std;
 7.
 8.  Triangulo::Triangulo() : Forma2D("Triangulo"), base(0.0),
 9.      altura(0.0)
10.  {
11.      // Cuerpo vacío
12.  }
13.
14.
15.  Triangulo::Triangulo(double unaBase, double unaAltura) :
16.      Forma2D("Triangulo")
17.  {
18.      establecerBase(unaBase);
19.      establecerAltura(unaAltura);
20.
```

```
21.  }
22.
23.
24.  Triangulo::Triangulo(string unTipo, double unaBase,
25.                        double unaAltura) : Forma2D(unTipo)
26.  {
27.      establecerBase(unaBase);
28.      establecerAltura(unaAltura);
29.
30.  }
31.
32.
33.  Triangulo::~Triangulo()
34.  {
35.      // Cuerpo vacío
36.  }
37.
38.
39.  void Triangulo::establecerBase(double unaBase)
40.  {
41.      // Validar la base
42.      if (unaBase >= 0.0)
43.      {
44.          base = unaBase;
45.      }
46.      else
47.      {
48.          base = 0.0;
49.
50.      }
51.
52.  }
53.
54.
55.  void Triangulo::establecerAltura(double unaAltura)
56.  {
57.      // Validar la altura
58.      if (unaAltura >= 0.0)
59.      {
60.          altura = unaAltura;
61.      }
62.      else
63.      {
64.          altura = 0.0;
65.
66.      }
67.
68.  }
69.
70.
71.  // Calcula el área del Triangulo
72.  double Triangulo::calcularArea()
73.  {
74.      return (obtenerBase() * obtenerAltura()) / 2.0;
75.
76.  }
```

```
77.
78.
79.  // Imprime un objeto de la clase Triangulo
80.  void Triangulo::imprimir()
81.  {
82.      Forma2D::imprimir();
83.
84.      cout << "Base: " << obtenerBase() << ", Altura: "
85.           << obtenerAltura() << ", Area: "
86.           << calcularArea() << endl;
87.
88.  }
```

4.4.4.7. La clase `Forma3D`

El Listado 25, es el archivo de cabecera de la clase `Forma3D`. La única variación importante, con relación a la implementación de la clase `Forma2D`, es la adición de la función miembro virtual pura `calcularVolumen()`. Como su nombre lo indica, esta función servirá para que las clases concretas derivadas de la clase `Forma3D` la redefinan a su particular conveniencia.

```
 1.  // Listado 25: forma3d.h
 2.  // Definición de la clase Forma3D
 3.  #ifndef FORMA3D_H
 4.  #define FORMA3D_H
 5.
 6.  #include <iostream>
 7.  #include "forma.h"
 8.
 9.
10.  class Forma3D : public Forma
11.  {
12.      public:
13.          Forma3D();
14.          explicit Forma3D(std::string unTipo);
15.
16.          virtual ~Forma3D();
17.
18.          virtual double calcularArea() = 0;
19.          virtual double calcularVolumen() = 0;
20.          virtual void imprimir() override;
21.      protected:
22.      private:
23.  };
24.
25.  #endif // FORMA3D_H
```

Y, la implementación de las funciones miembros de esta clase `Forma3D`, se encuentra en el Listado 26:

```
 1.  // Listado 26: forma3d.cpp
 2.  // Definición de las funciones miembro de la clase Forma3D
 3.  #include <iostream>
 4.  #include "forma3d.h"
 5.
 6.  using namespace std;
 7.
 8.
 9.  Forma3D::Forma3D() : Forma("Forma3D")
10.  {
11.      // Cuerpo vacío
12.  }
13.
14.
15.  Forma3D::Forma3D(std::string unTipo) : Forma(unTipo)
16.  {
17.      // Cuerpo vacío
18.  }
19.
20.
21.  Forma3D::~Forma3D()
22.  {
23.      // Cuerpo vacío
24.  }
25.
26.
27.  // Imprime en la pantalla un objeto de la clase Forma3D
28.  void Forma3D::imprimir()
29.  {
30.      Forma::imprimir();
31.
32.  }
```

4.4.4.8. Clases concretas derivadas de la clase `Forma3D`

El listado siguiente, el número 27, nos enseña el archivo de cabecera con la definición de la clase `Esfera`, derivada de la clase `Forma3D` y la primera de las clases concretas que implementan todas las funciones virtuales de las clases base `Forma` y `Forma3D`. Las otras son la clase `Cubo` y la clase `Tetraedro`.

```
1.  // Listado 27: esfera.h
2.  // Definición de la clase Esfera
3.  #ifndef ESFERA_H
4.  #define ESFERA_H
5.
6.  #include <iostream>
```

```
 7.  #include "forma3d.h"
 8.
 9.
10.  class Esfera : public Forma3D
11.  {
12.      public:
13.          Esfera();
14.          explicit Esfera(double unRadio);
15.          Esfera(std::string unTipo, double unRadio);
16.
17.          virtual ~Esfera();
18.
19.          double obtenerRadio() { return radio; }
20.          void establecerRadio(double unRadio);
21.
22.          virtual double calcularArea() override;
23.          virtual double calcularVolumen() override;
24.          virtual void imprimir() override;
25.      protected:
26.      private:
27.          double radio;
28.  };
29.
30.  #endif // ESFERA_H
```

La implementación de las funciones miembro correspondientes se presenta en el Listado 28.

```
 1.  // Listado 28: esfera.cpp
 2.  // Definición de las funciones miembro de la clase Esfera
 3.  #include <iostream>
 4.  #include <cmath>
 5.  #include "esfera.h"
 6.
 7.  using namespace std;
 8.
 9.  const double PI = 3.141592653589793;
10.
11.
12.  Esfera::Esfera() : Forma3D("Esfera"), radio(0.0)
13.  {
14.      // Cuerpo vacío
15.  }
16.
17.
18.  Esfera::Esfera(double unRadio) : Forma3D("Esfera")
19.  {
20.      establecerRadio(unRadio);
21.  }
22.
23.
24.  Esfera::Esfera(string unTipo, double unRadio) : Forma3D(unTipo)
25.  {
26.      establecerRadio(unRadio);
```

```
27.  }
28.
29.
30.  Esfera::~Esfera()
31.  {
32.      // Cuerpo vacío
33.  }
34.
35.
36.  void Esfera::establecerRadio(double unRadio)
37.  {
38.      // Validar el radio
39.      if (unRadio >= 0.0)
40.      {
41.          radio = unRadio;
42.      }
43.      else
44.      {
45.          radio = 0.0;
46.
47.      }
48.
49.  }
50.
51.
52.  // Calcula el área de la esfera
53.  double Esfera::calcularArea()
54.  {
55.      return PI * obtenerRadio() * obtenerRadio() * 4;
56.
57.  }
58.
59.
60.  // Calcula el volumen de la esfera
61.  double Esfera::calcularVolumen()
62.  {
63.      double r = obtenerRadio();
64.
65.      return PI * r * r * r * 4 / 3;
66.
67.  }
68.
69.
70.  // Imprime un objeto de la clase Esfera
71.  void Esfera::imprimir()
72.  {
73.      Forma3D::imprimir();
74.
75.      cout << "Radio: " << obtenerRadio() << ", Area: "
76.           << calcularArea() << endl;
77.
78.  }
```

Como se puede observar, la similitud de la implementación de las clases derivadas de la clase `Forma3D`, con relación a las contrapartes implementadas para la clase `Forma2D`, es muy alta. Por este motivo, dejaremos como ejercicio al lector que analice el código fuente de cada uno de estos listados, comparándolos contra los que analizamos paso por paso, con la finalidad de que reafirme lo aprendido.

```
 1.  // Listado 29: cubo.h
 2.  // Definición de la clase Cubo
 3.  #ifndef CUBO_H
 4.  #define CUBO_H
 5.
 6.  #include <iostream>
 7.  #include "forma3d.h"
 8.
 9.
10.  class Cubo : public Forma3D
11.  {
12.      public:
13.          Cubo();
14.          explicit Cubo(double unLado);
15.          Cubo(std::string unTipo, double unLado);
16.
17.          virtual ~Cubo();
18.
19.          double obtenerLado() { return lado; }
20.          void establecerLado(double unLado);
21.
22.          virtual double calcularArea() override;
23.          virtual double calcularVolumen() override;
24.          virtual void imprimir() override;
25.      protected:
26.      private:
27.          double lado;
28.  };
29.
30.  #endif // CUBO_H
```

Y la implementación de funciones está dado por:

```
 1.  // Listado 30: cubo.cpp
 2.  // Definición de las funciones miembro de la clase Cubo
 3.  #include <iostream>
 4.  #include "cubo.h"
 5.
 6.  using namespace std;
 7.
 8.
 9.  Cubo::Cubo() : Forma3D("Cubo"), lado(0.0)
10.  {
```

```
11.      // Cuerpo vacío
12.  }
13.
14.
15.  Cubo::Cubo(double unLado) : Forma3D("Cubo")
16.  {
17.      establecerLado(unLado);
18.
19.  }
20.
21.
22.  Cubo::Cubo(string unTipo, double unLado) :
23.      Forma3D(unTipo)
24.  {
25.      establecerLado(unLado);
26.
27.  }
28.
29.
30.  Cubo::~Cubo()
31.  {
32.      // Cuerpo vacío
33.
34.  }
35.
36.
37.  void Cubo::establecerLado(double unLado)
38.  {
39.      // Validar el lado
40.      if (unLado >= 0.0)
41.      {
42.          lado = unLado;
43.      }
44.      else
45.      {
46.          lado = 0.0;
47.
48.      }
49.
50.  }
51.
52.
53.  // Calcula el área del Cubo
54.  double Cubo::calcularArea()
55.  {
56.      return 6 * obtenerLado() * obtenerLado();
57.
58.  }
59.
60.
61.  // Calcula el volumen del Cubo
62.  double Cubo::calcularVolumen()
63.  {
64.      return obtenerLado() * obtenerLado() * obtenerLado();
65.
66.  }
```

```
67.
68.
69.  // Imprime un objeto de la clase Cubo
70.  void Cubo::imprimir()
71.  {
72.      Forma3D::imprimir();
73.
74.      cout << "Lado: " << obtenerLado() << ", Area: "
75.           << calcularArea() << endl;
76.
77.  }
```

Las diferencias obvias son, por supuesto, como la implementación de la función miembro `calcularVolumen()`, para lo cual se aplican las fórmulas matemáticas de geometría básica. Además, en algunos de los listados se incluye el archivo de cabecera "`<cmath>`", el cual contiene los prototipos de las funciones de la librería matemática, lo que nos permite llamar a las funciones como `sqrt()`, que calcula la raíz cuadrada de un número y que se emplea en el cálculo de algunas áreas y volúmenes de las formas geométricas tridimensionales que estamos implementando.

```
 1.  // Listado 31: tetraedro.h
 2.  // Definición de la clase Tetraedro
 3.  #ifndef TETRAEDRO_H
 4.  #define TETRAEDRO_H
 5.
 6.  #include <iostream>
 7.  #include "forma3d.h"
 8.
 9.
10.  class Tetraedro : public Forma3D
11.  {
12.      public:
13.          Tetraedro();
14.          explicit Tetraedro(double unLadoTriangulo);
15.          Tetraedro(std::string unTipo, double unLadoTriangulo);
16.
17.          virtual ~Tetraedro();
18.
19.          double obtenerLadoTriangulo() { return ladoTriangulo; }
20.          void establecerLadoTriangulo(double unLadoTriangulo);
21.
22.          virtual double calcularArea() override;
23.          virtual double calcularVolumen() override;
24.          virtual void imprimir() override;
25.      protected:
26.      private:
27.          double ladoTriangulo;
28.  };
29.
```

```
30.  #endif // TETRAEDRO_H
```

Y, finalmente, la implementación se encuentra en el Listado 32.

```
 1.  // Listado 32: tetraedro.cpp
 2.  // Definición de las funciones miembro de la clase Tetraedro
 3.  #include <iostream>
 4.  #include <cmath>
 5.  #include "tetraedro.h"
 6.
 7.  using namespace std;
 8.
 9.
10.  Tetraedro::Tetraedro() : Forma3D("Tetraedro"), ladoTriangulo(0.0)
11.  {
12.      // Cuerpo vacío
13.  }
14.
15.
16.  Tetraedro::Tetraedro(double unLadoTriangulo) :
17.      Forma3D("Tetraedro")
18.  {
19.      establecerLadoTriangulo(unLadoTriangulo);
20.
21.  }
22.
23.
24.  Tetraedro::Tetraedro(string unTipo, double unLadoTriangulo) :
25.      Forma3D(unTipo)
26.  {
27.      establecerLadoTriangulo(unLadoTriangulo);
28.
29.  }
30.
31.
32.  Tetraedro::~Tetraedro()
33.  {
34.      // Cuerpo vacío
35.
36.  }
37.
38.
39.  void Tetraedro::establecerLadoTriangulo(double unLadoTriangulo)
40.  {
41.      // Validar el ladoTriangulo
42.      if (unLadoTriangulo >= 0.0)
43.      {
44.          ladoTriangulo = unLadoTriangulo;
45.      }
46.      else
47.      {
48.          ladoTriangulo = 0.0;
49.
50.      }
51.
```

```
52.  }
53.
54.
55.  // Calcula el área del Tetraedro
56.  double Tetraedro::calcularArea()
57.  {
58.      return sqrt(3.0) * obtenerLadoTriangulo() *
59.          obtenerLadoTriangulo();
60.  }
61.
62.
63.  // Calcula el volumen del Tetraedro
64.  double Tetraedro::calcularVolumen()
65.  {
66.      double lado = obtenerLadoTriangulo();
67.
68.      return sqrt(2.0) * lado * lado * lado / 12.0;
69.
70.  }
71.
72.
73.  // Imprime un objeto de la clase Tetraedro
74.  void Tetraedro::imprimir()
75.  {
76.      Forma3D::imprimir();
77.
78.      cout << "ladoTriangulo: " << obtenerLadoTriangulo()
79.           << ", Area: " << calcularArea() << endl;
80.
81.  }
```

4.4.5. Polimorfismo

La pieza faltante del rompecabezas, para el procesamiento polimórfico, nos la da el Listado 33, en el cual se muestra el programa que pone a prueba todas las clases anteriormente desarrolladas sobre formas geométricas.

En las líneas 4 a 15 de este Listado 33, se incluyen todos los archivos de cabecera necesarios para acceder a las definiciones de las clases en la jerarquía de herencia de las formas geométricas. Posteriormente, en las líneas 22 a 35, declaramos un arreglo de variables de tipo apuntador a objetos de la clase `Forma`, la clase base de toda la jerarquía de herencia que hemos estado desarrollando. Al mismo tiempo, instanciamos dinámicamente dos objetos del tipo de cada una de las seis clases

concretas (`Circulo`, `Cuadrado`, `Triangulo`, `Esfera`, `Cubo` y `Tetraedro`) derivadas de las clases `Forma2D` y `Forma3D` derivadas, a su vez, de la clase `Forma`. Estos doce objetos instanciados dinámicamente, mediante el operador `new`, los colocamos en el arreglo mencionado. La posibilidad de asignar un apuntador a un objeto de una clase derivada (tipo `Cuadrado*`, por ejemplo) a un apuntador a un objeto de una clase base (tipo `Forma*`, por ejemplo), es uno de los fundamentos del polimorfismo, el otro son las funciones miembro virtuales de clases base abstractas.

```
 1.  // Listado 33: pruebapolimorfismo.cpp
 2.  // Programa de prueba del procesamiento polimórfico de objetos
 3.  // de la clase Forma
 4.  #include <iostream>
 5.  #include <iomanip>
 6.  #include "colores.h"
 7.  #include "forma.h"
 8.  #include "forma2d.h"
 9.  #include "circulo.h"
10.  #include "cuadrado.h"
11.  #include "triangulo.h"
12.  #include "forma3d.h"
13.  #include "esfera.h"
14.  #include "cubo.h"
15.  #include "tetraedro.h"
16.
17.  using namespace std;
18.
19.
20.  int main()
21.  {
22.      Forma *formas[] = {
23.          new Circulo("Circulo", 2.0),
24.          new Circulo("Circulo", 1.5),
25.          new Cuadrado("Cuadrado", 2.5),
26.          new Cuadrado("Cuadrado", 3.0),
27.          new Triangulo("Triangulo", 4.0, 2.5),
28.          new Triangulo("Triangulo", 3.0, 2.25),
29.          new Esfera("Esfera", 6.0),
30.          new Esfera("Esfera", 4.5),
31.          new Cubo("Cubo", 2.0),
32.          new Cubo("Cubo", 4.0),
33.          new Tetraedro("Tetraedro", 3.5),
34.          new Tetraedro("Tetraedro", 8.0)
35.      };
36.
37.      // Establecer color de contorno y relleno para cada forma
38.
39.      // Formas 2D
40.
41.      formas[0]->establecerColorContorno(SIN_COLOR);
42.      formas[0]->establecerColorRelleno(AMARILLO);
43.
```

```
44.      formas[1]->establecerColorContorno(NEGRO);
45.      formas[1]->establecerColorRelleno(BLANCO);
46.
47.      formas[2]->establecerColorContorno(AZUL);
48.      formas[2]->establecerColorRelleno(ROJO);
49.
50.      formas[3]->establecerColorContorno(CIAN);
51.      formas[3]->establecerColorRelleno(ROSA);
52.
53.      formas[4]->establecerColorContorno(GRIS_OBSCURO);
54.      formas[4]->establecerColorRelleno(NARANJA);
55.
56.      formas[5]->establecerColorContorno(GRIS);
57.      formas[5]->establecerColorRelleno(MAGENTA);
58.
59.      // Formas 3D
60.
61.      formas[6]->establecerColorContorno(VERDE);
62.      formas[6]->establecerColorRelleno(GRIS_CLARO);
63.
64.      formas[7]->establecerColorContorno(GRIS_CLARO);
65.      formas[7]->establecerColorRelleno(VERDE);
66.
67.      formas[8]->establecerColorContorno(MAGENTA);
68.      formas[8]->establecerColorRelleno(GRIS);
69.
70.      formas[9]->establecerColorContorno(NARANJA);
71.      formas[9]->establecerColorRelleno(GRIS_OBSCURO);
72.
73.      formas[10]->establecerColorContorno(ROSA);
74.      formas[10]->establecerColorRelleno(CIAN);
75.
76.      formas[11]->establecerColorContorno(ROJO);
77.      formas[11]->establecerColorRelleno(AZUL);
78.
79.      // Configurar la impresión de números de punto flotante
80.      cout << fixed << setprecision(2);
81.
82.      // Procesar en forma polimórfica los objetos de la clase Forma
83.      for(Forma *forma : formas)
84.      {
85.          forma->imprimir();
86.          cout << endl;
87.
88.          // Intentar convertir el apuntador a un objeto de la clase
89.          // Forma a apuntador a objeto de la clase Forma3D (para
90.          // calcular el volumen)
91.
92.          Forma3D *forma3d = dynamic_cast<Forma3D *>(forma);
93.
94.          if (forma3d != nullptr) // Verdadero para una relación es-un
95.          {
96.              double volumen = forma3d->calcularVolumen();
97.
98.              cout << "... y el volumen es : " << volumen << "\n"
99.                   << endl;
```

```
100.
101.           } // if
102.
103.       } // foreach forma
104.
105.       return 0;
106.
107.   } // main()
```

4.4.5.1. Relaciones entre objetos en una jerarquía de herencia

En el capítulo anterior, mencionábamos que el polimorfismo significa que los objetos que son instancias de clases diferentes, pero todas ellas con una clase base en común, pueden proveer un servicio como respuesta al mismo mensaje, es decir, cada objeto responde a su manera. En este sentido, teniendo en C++ una variable de tipo apuntador a un objeto de una clase base y una variable de tipo apuntador a un objeto de una clase derivada, podemos realizar dos operaciones con ellos, apuntarlos a un objeto que realmente es del tipo de la clase base, o apuntarlos a un objeto que realmente es del tipo de la clase derivada. Para ejemplificar de mejor manera lo anterior, pensemos que la clase base es la clase `Forma` que hemos desarrollado en apartados anteriores y que la clase derivada es la clase `Cuadrado`, también vista previamente. En consecuencia, podemos ver en la Figura 11 la representación gráfica de las cuatro operaciones mencionadas:

a) Apuntamos a un objeto `Forma` el apuntador de tipo `Forma*`.
b) Apuntamos a un objeto `Forma` el apuntador de tipo `Cuadrado*`.
c) Apuntamos a un objeto `Cuadrado` el apuntador de tipo `Forma*`.
d) Apuntamos a un objeto `Cuadrado` el apuntador de tipo `Cuadrado*`.

Claramente, los incisos a) y d) no tienen mayor problema, puesto que estamos asignando a un apuntador de un tipo, un objeto de ese mismo tipo. Ahora bien, el inciso b) generará errores de compilación, ya que pretendemos asignar la dirección de un objeto de tipo `Forma` a un apuntador a un objeto de tipo `Cuadrado`. El problema viene del hecho de que este último apuntador supone que el objeto al que apunta cuenta no solo con las características de la propia clase `Cuadrado`, sino

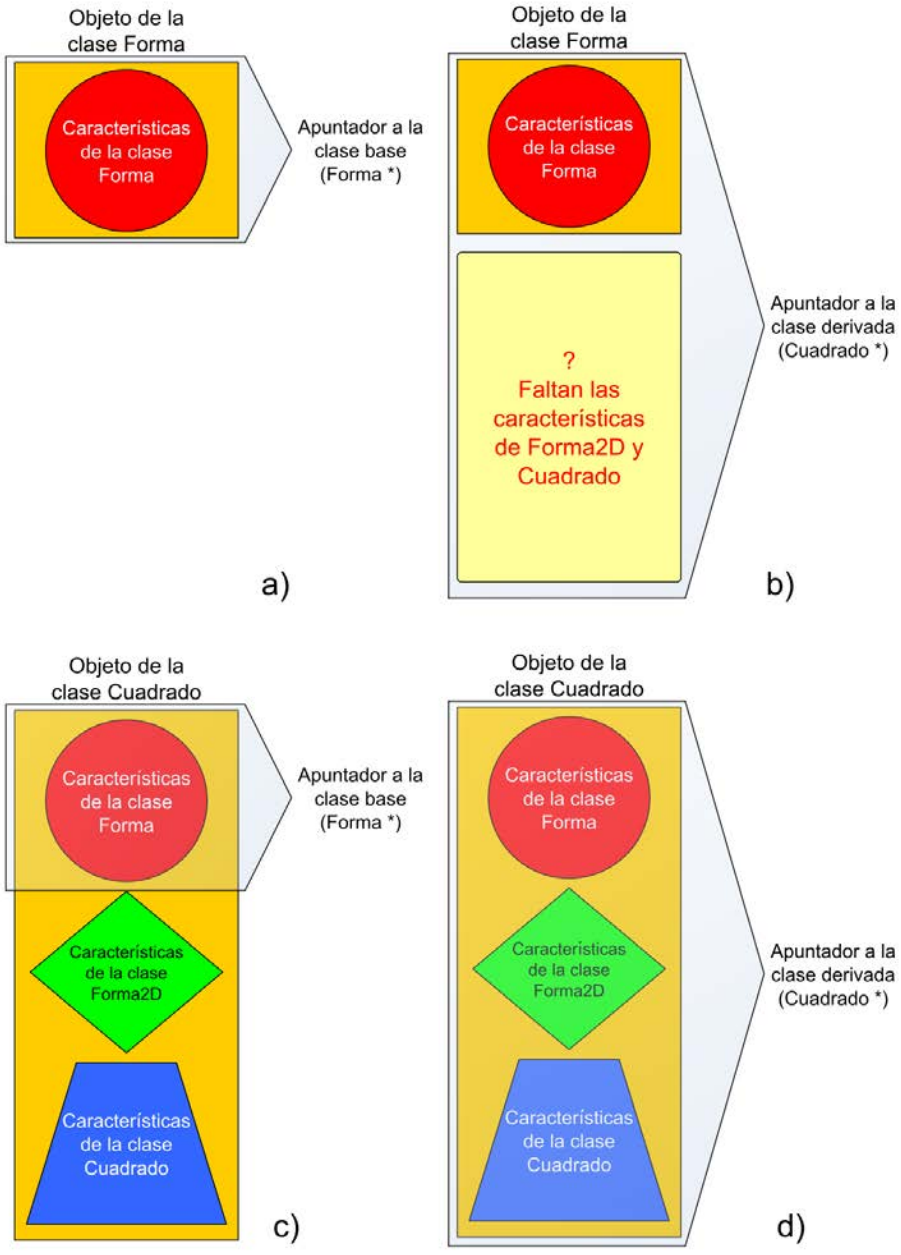

Figura 11. Relaciones entre objetos en una jerarquía de herencia

también de la clase `Forma` y de la clase `Forma2D`, las cuales por supuesto no tiene.

En último lugar, el inciso d) no tendrá problema, puesto que el apuntador a un objeto de tipo `Forma`, solo accesará la parte correspondiente a las características de la clase `Forma`, parte con la que sí cuenta el objeto real (además de las características de la clase `Forma2D` y `Cuadrado`, claro está).

En este sentido, las líneas 22 a 35 del Listado 33, realizan precisamente lo que el inciso d) muestra, asignar a un apuntador a un objeto de una clase base un apuntador a un objeto de una clase derivada. Posteriormente, en las líneas 37 a 77, hacemos uso de la parte de las características de la clase `Forma` que los objetos en el arreglo tienen, llamando a las funciones miembro que establecen el color del contorno y el color de relleno para asignarle a cada forma dichos colores.

En las líneas 79 y 80 se configura la impresión de números de punto flotante en notación fija y con dos dígitos después del punto decimal.

Finalmente llegamos a la parte del procesamiento polimórfico, líneas 83 a 103, en la cual utilizamos un ciclo `for` especial que C++ proporciona para recorrer conjuntos de elementos, como un arreglo. Un ejemplo más sencillo del uso de este `for` mejorado es el siguiente:

```
int calificaciones[5] = { 10, 9, 10, 7, 10 };
double suma = 0.0;
for (int calificacion : calificaciones)
{
    suma += calificacion;
}
double promedio = suma / 5.0;
cout << "El promedio es: " << promedio << endl;
```

En este ejemplo pequeño, el ciclo `for` mejorado recorrerá el arreglo `calificaciones`, asignando cada uno de sus elementos, uno a la vez, a la variable `calificacion`, la cual se procesará individualmente dentro del ciclo, añadiendo su valor a la variable `suma`. Al final del programa, se calcula el valor promedio del arreglo y se imprime en pantalla.

De igual manera que para el arreglo `calificaciones`, el ciclo `for` mejorado de las líneas 83 a 103, recorrerá el arreglo `formas`, asignando cada uno de sus elementos, uno a la vez, a la variable `forma`, la cual se procesará individualmente dentro del ciclo.

Dentro del ciclo `for` mejorado, la variable `forma` se emplea para llamar a la función `imprimir()` de la parte de las características de la clase `Forma` que todos los objetos de las clases derivadas tienen.

4.4.5.2. Utilidad de las funciones `virtual`

Si nos quedáramos hasta aquí, la función `imprimir()` de la clase `Forma` solo mostraría sus variables miembro para cada uno de los objetos en el arreglo. Lo cual no es lo que queremos. La diferencia viene cuando recordamos que en la definición de la clase `Forma` (Listado 15), la función miembro `imprimir()` es una función `virtual`. Esta palabra reservada le indica al compilador de C++ que, cuando una clase derivada de la clase `Forma` redefina la función miembro `imprimir()`, deberá emplear esa definición, y no la de clase base, cuando intentemos ejecutar la función `imprimir()` de la clase base (como en la línea 85 del Listado 33). Esto hace que la función miembro `imprimir()` llamada, sea la específica de cada clase derivada, por lo tanto, la información que se imprime es la que cada clase derivada deseé mostrar.

4.4.6 Información de tiempo de ejecución (*RTTI – RunTime Type Information*) y coerción de tipos descendente dinámica[2]

En esta sección presentamos una de las muchas mejoras que el estándar C++11 incluye en el lenguaje. En las líneas 88 a 101 del Listado 33, queremos determinar si el objeto que estamos procesando mediante al apuntador `forma` es del tipo de una clase derivada de la clase `Forma3D`, esto con la finalidad de calcular el volumen de dicha forma geométrica. Si el objeto no es del tipo `Forma3D` (recordemos que un objeto de una clase derivada *es-un* objeto de su clase base) no tendría ni declarada ni definida la función `calcularVolumen()`. Para tal fin, en la línea 92 usamos el operador `dynamic_cast` para intentar la coerción del tipo del apuntador `forma`, de modo descendente y dinámico.[vii] Luego, en la línea 94 verificamos el apuntador devuelto por el operador `dynamic_cast`, si este no es `nullptr`[viii] significa que el objeto al que apunta `forma` es un objeto de tipo `Forma3D` y podemos, de modo seguro, llamar a la función `calcularVolumen()` que, empleando polimorfismo nuevamente, ejecutará la función miembro específica de cada objeto.

Ya solo resta devolver un indicador de éxito al sistema operativo, línea 105 del Listado 33, y finalizar la ejecución del programa.

Por último, la salida del programa del Listado 33 es la siguiente:

```
Objeto de tipo: Circulo.
Color del contorno: Ninguno.
```

[vii] La coerción es *descendente*, porque intentamos convertir un apuntador a un objeto de tipo `Forma`, a un apuntador a un objeto de tipo `Forma3D`, y la clase `Forma3D` se deriva de la clase `Forma` (descendemos en el árbol de la jerarquía de herencia). Y es *dinámica*, porque la coerción de tipo se intenta realizar al momento de ejecutar el programa, no cuando lo estamos compilando.

[viii] `nullptr` es otra de las adiciones al lenguaje en el estándar C++11, es una constante que indica que un apuntador *apunta a nada*, pero más segura que la constante `NULL` a la que los programadores del lenguaje C están acostumbrados.

```
Color del relleno: Amarillo.
Radio: 2.00, Area: 12.57


Objeto de tipo: Circulo.
Color del contorno: Negro.
Color del relleno: Blanco.
Radio: 1.50, Area: 7.07


Objeto de tipo: Cuadrado.
Color del contorno: Azul.
Color del relleno: Rojo.
Lado: 2.50, Area: 6.25


Objeto de tipo: Cuadrado.
Color del contorno: Cian.
Color del relleno: Rosa.
Lado: 3.00, Area: 9.00


Objeto de tipo: Triangulo.
Color del contorno: Gris obscuro.
Color del relleno: Naranja.
Base: 4.00, Altura: 2.50, Area: 5.00


Objeto de tipo: Triangulo.
Color del contorno: Gris.
Color del relleno: Magenta.
Base: 3.00, Altura: 2.25, Area: 3.38


Objeto de tipo: Esfera.
Color del contorno: Verde.
Color del relleno: Gris claro.
Radio: 6.00, Area: 452.39
```

```
... y el volumen es : 904.78


Objeto de tipo: Esfera.
Color del contorno: Gris claro.
Color del relleno: Verde.
Radio: 4.50, Area: 254.47

... y el volumen es : 381.70


Objeto de tipo: Cubo.
Color del contorno: Magenta.
Color del relleno: Gris.
Lado: 2.00, Area: 24.00

... y el volumen es : 8.00


Objeto de tipo: Cubo.
Color del contorno: Naranja.
Color del relleno: Gris obscuro.
Lado: 4.00, Area: 96.00

... y el volumen es : 64.00


Objeto de tipo: Tetraedro.
Color del contorno: Rosa.
Color del relleno: Cian.
ladoTriangulo: 3.50, Area: 21.22

... y el volumen es : 5.05


Objeto de tipo: Tetraedro.
Color del contorno: Rojo.
```

```
Color del relleno: Azul.
ladoTriangulo: 8.00, Area: 110.85

... y el volumen es : 60.34
```

Referencias

[1] Deitel, P. & Deitel, H. (2014) C++ How to Program, 9th. Ed. Boston, Massachusetts: Pearson Education, Inc.

[2] Amdocs. "Object Oriented Concepts." Consultado: 29 de noviembre de 2013, de http://pro-cess.co.il/downloads/Dreams_come_true.ppt