Capítulo 1

Programación Orientada a Objetos

1.1. Paradigmas de programación

Haremos un breve repaso por los principales paradigmas de la programación según su avance cronológico.

1.1.1. Programación no estructurada

En un principio, los programadores no conservaban ningún estilo, cada uno programaba a su gusto personal sin un criterio unificado. Algunos lenguajes, como el Basic, permitían "saltos" de una instrucción a otra, por medio de sentencias como *goto* y *exit*. Estas características hacían que los programas fueran muy difíciles de corregir y mantener. Si se detectaba algún error en la ejecución de un programa se debía revisar el código de forma completa para poder determinar dónde estaba el problema.

Por otro lado, con respecto al mantenimiento, actualizar un programa, por ejemplo, por nuevas normativas impositivas o agregarle nueva funcionalidad para obtener algún reporte que hasta el momento no se generaba; era muy complejo. Tomar un programa que, probablemente, había escrito otra persona y tratar de entender y seguir el hilo de la ejecución era una tarea prácticamente imposible de realizar.

1.1.2. Programación estructurada

A partir de la década del 70 se comenzó a implementar la programación estructurada, basada en el Teorema de Bohm y Jacopini, del año 1966, en donde se determinaba que cualquier algoritmo podía ser resuelto mediante tres estructuras básicas:

- Secuenciales
- Condicionales
- Repetitivas

Estructuras que se vieron en materias anteriores. En la programación estructurada se prohíbe el uso del *goto* y otras sentencias de ruptura con la finalidad de lograr un código más legible y encontrar rápidamente los errores sin necesidad de estar revisando el código por completo.

La idea de la programación estructurada es resolver los problemas de forma *top–down*, es decir, de arriba hacia abajo, con el objetivo de ir particionando un problema complejo en varios más simples.

El programa principal estará formado por la llamada a las subrutinas (funciones) más importantes, las cuales, a su vez, llamarán a otras subrutinas y, así sucesivamente, hasta que cada subrutina solo se dedique a realizar una tarea sencilla. Por ejemplo, si necesitáramos tomar datos de un archivo A con los que se cargaría una tabla, luego mostrar los resultados, a continuación tomar datos desde el teclado y actualizar el archivo A; nuestro programa principal podría tener la siguiente forma:

```
// Programa en pseudocodigo
Principal
Comienzo
    abrir_archivo(A);
    cargar_tabla(A, tabla, n); // n: tamaño de tabla
    mostrar_resultado(tabla, n);
    tomar_datos(tabla, n);
    actualizar_archivo(A, tabla, n);
Fin.
```

Hay que tener en cuenta que primero se arma el esqueleto del programa y, luego, se va rellenando. Es decir, se van escribiendo las funciones que hagan falta.

1.1.3. Programación orientada a objetos

La necesidad de extender programas (agregar funcionalidad) y reutilizar código (no volver a escribir lo mismo dos o más veces), hizo que el paradigma de la programación estructurada evolucionara hacia un nuevo paradigma: la Programación Orientada a Objetos.

Este paradigma intenta modelar el mundo real en el cual nos encontramos con toda clase de objetos que se comunican entre sí y reaccionan ante determinados estímulos o mensajes. Los objetos están conformados por propiedades o atributos, incluso, estos atributos pueden ser, a su vez, otros objetos. Y tienen cierto comportamiento, es decir, ante un determinado estímulo reaccionan de una determinada manera. Además, en un momento preciso tienen un estado que está definido por el valor de sus atributos. Por ejemplo, el objeto semáforo tiene luces (sus atributos), que reaccionan ante el paso del tiempo. Cada cierta cantidad de segundos, estas luces, cambian de estado: de prendido a apagado o viceversa. Es decir, en un determinado momento su estado será:

- Luz roja prendida.
- Luz amarilla apagada.
- Luz verde apagada.

Cuando pasen x segundos, recibirá un mensaje *prender_luz_amarilla*. En cambio, el objeto *auto* no reaccionará ante el paso del tiempo, sino que reaccionará ante los estímulos de la persona que lo maneje, como encender, acelerar, frenar, etc. El objeto *auto* tendrá como atributos otros objetos, como *radio*, *control_remoto*, *puerta*, etc.

El objeto *celular_1* recibirá una señal de otro objeto que es el objeto *antena* y, a través de éste, se comunicará con otro objeto *celular_2* que, seguramente, tendrá otras características: otro tamaño, marca, etc. Sin embargo, aunque estos celulares sean muy distintos, ambos pertenecerán a una misma clase: la clase *Celular* y se comunicarán mediante una interfaz común a ambos.

¿Cómo pensar en POO?

¿Cómo se deben encarar los problemas con este paradigma? En la programación estructurada se pensaba la solución de un problema de la forma *top – down*, es decir, de arriba hacia abajo, llamando a funciones que fueran dividiendo nuestro problema en uno más pequeño, hasta encontrarnos con problemas muy sencillos.

En la POO la forma de resolver un problema es inversa. En lugar de determinar qué funciones necesitamos se debe determinar qué objetos se necesitarán. Los problemas se irán resolviendo de abajo hacia arriba, desde pequeños objetos que actuarán entre sí y conformarán otros más complejos.

Por ejemplo, si quisiéramos cargar un vector y, luego, ordenarlo y listarlo, en Programación Estructurada pensaríamos en las siguientes funciones:

```
// Estructurado
cargar(vector, n); // Funcion que carga el vector
ordenar(vector, n); // Funcion que ordena al vector
mostrar(vector, n); // Funcion que muestra al vector
```

Estas funciones tendrían, como parámetros, el vector que deseamos cargar, ordenar y listar; y un valor entero n, su tamaño. Hay que notar que, por ejemplo, la función ordenar no trabajaría de la misma forma en un vector de enteros que de strings, por lo que deberíamos escribir una nueva función para las distintas clases de vectores. Este era uno de los inconvenientes que mencionamos en la Programación Estructurada: no hay reutilización del código, ya que estos procedimientos serían prácticamente los mismos, solo que cambiarían sus tipos de datos. Más adelante veremos programación genérica y polimorfismo, dos formas de solucionar este problema.

La misma situación, pero encarada en POO, sería de la siguiente manera: en primer lugar se debe establecer una clase *Vector* que, tendrá como datos, los valores que se desean almacenar y el tamaño del vector. Además, tendrá métodos para cargarlos, ordenarlos y mostrarlos:

```
// POO
Vector v; // Se crea un objeto de tipo Vector
v.cargar(); // Se llama al metodo cargar
v.ordenar(); // Se llama al metodo ordenar
```

```
v.mostrar(); // Se llama al metodo mostrar
```

En el código anterior vemos que no hay necesidad de pasar el vector por parámetro ya que es el propio objeto que llama a los métodos. Tampoco necesitamos pasar su tamaño como en la programación estructurada porque el propio vector sabe cuál es su tamaño.

1.2. Tipo Abstracto de Datos

El lector se encontrará familiarizado con la expresión *tipo de dato* ya que habrá escrito una serie de programas o aplicaciones en las que utilizó diferentes tipos de datos, como un entero para guardar una edad o una cadena de caracteres para almacenar un nombre. Sin embargo, el término *abstracción*, probablemente, le resulte nuevo en el ámbito de la informática. Quizá esa palabra le represente una imagen de alguna canción de Spinetta. También, podría recordarle alguna frase peyorativa de su maestra de primaria "estás abstraído", refiriéndose a que no le prestaba atención. Estas imágenes no tienen que ver con la idea que se le quiere dar al término *abstracción* en informática.

El diccionario de la Real Academia Española indica:

- Abstracción. Acción y efecto de abstraer o abstraerse.
- Abstraer. Separar, por medio de una operación intelectual, las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción.

Si bien esto no nos aclara demasiado ni da un indicio de cómo se puede utilizar en el desarrollo de alguna aplicación, nos da la idea que debemos separar las cualidades de un objeto para quedarnos con su esencia.

En el cuento *Funes el memorioso*, Jorge Luis Borges describe a una persona (Funes) que recuerda todo lo que ha vivido con la mayor exactitud. El narrador dice en un párrafo "Sospecho, sin embargo, que no era muy capaz de pensar. Pensar es olvidar diferencias, es generalizar, abstraer." Esta es la clave del término *abstracción*, si no podemos abstraernos no podremos pensar, por lo tanto, tampoco actuar.

Imaginemos, por un momento, que venimos de un planeta donde no existen perros ni gatos. Al llegar a la tierra nos ponemos a charlar con un terrícola y vemos pasar, por al lado nuestro, un Gran Danés. Al interrogar sobre qué clase de animal es, el terrícola nos contesta que es un perro. Un rato más tarde vemos un Caniche. No podemos imaginarnos que, también, es un perro. Porque su tamaño, el color, el pelaje es completamente distinto al anterior. Sin embargo, el terrícola insiste en que eso que estamos viendo y que emite unos ladridos agudos, no es un juguete, sino un perro. A los pocos minutos cruza un gato por nuestro camino. Le preguntamos al terrícola si, también, es un perro, porque no tiene tanta diferencia con el Caniche. Sin embargo, el terrícola, riéndose, nos dice que es un gato, que no tiene nada que ver con un perro. Diariamente nos cruzamos con perros que jamás hemos visto. Sin embargo logramos reconocerlos. ¿Por qué? Porque de chicos preguntamos miles de veces a los mayores "¿qué es eso?" y nos contestaban que era un perro. Luego de ver decenas de ellos, encontramos, inconscientemente, una regla que nos indica "si lo que ves es así, así y así, entonces, es un perro". Es decir, nos quedamos con la esencia de las cosas, las simplificamos, nos abstraemos.

Por otro lado, es importante que tengamos la capacidad de utilizar las cosas sin saber cómo están implementadas, sabiendo solo cómo reaccionarán ante determinado estímulo. Para comprender esto se darán algunos ejemplos.

Cuando giramos la llave de contacto de un automóvil sabemos que se pondrá en marcha y, luego de colocar el cambio adecuado y acelerar, se pondrá en movimiento. Sin embargo, no necesitamos saber qué mecanismos internos se están accionando en cada instante ni qué piezas del motor se ponen en actividad. Por más que lo sepamos, no estaremos pensando en cada detalle de lo que está sucediendo "por debajo" de la interfaz. Por interfaz nos referimos al sistema que se comunica con el usuario, en este caso, llave de contacto, palanca de cambios, volante, acelerador, etc. El usuario sería el conductor. Estar pendientes de cada pieza que actúa en el automóvil al momento de estar conduciéndolo, sólo logrará distraernos y entorpecer nuestro objetivo principal que es el de conducir hacia un determinado lugar.

Un médico, cuando camina, no está pendiente de las órdenes que envía el cerebro a determinados músculos a través del sistema nervioso para que se pongan

en movimiento y logre avanzar algunos pasos, aunque lo haya estudiado en alguna materia.

Con respecto a los tipos de datos abstractos (TDA), en informática, nos encontraremos principalmente con tres enfoques distintos:

- Diseñador. Como diseñador haremos uno o varios diseños del tipo de dato a construir. En el ejemplo de los automóviles haríamos planos y gráficos, y algún documento indicando cómo será el automóvil a fabricar, qué cosas tendrá, etc.

- Implementador. Como implementadores construiremos nuevos tipos de datos abstractos para que otros los utilicen (podríamos ser nosotros mismos). Siguiendo el ejemplo del automóvil, en este papel, seríamos la fábrica de automóviles. Como fábrica desearemos hacer automóviles fuertes, seguros, sencillos de utilizar. Por otro lado, no desearemos que los usuarios "metan mano" en los dispositivos construidos, por lo que se tomarán los recaudos necesarios para que esto no suceda como fajas de seguridad, etc. Hay que tener en cuenta que la garantía de una máquina se pierde en caso de detectarse que una persona no autorizada estuvo modificando la implementación.

- Usuario. Como usuarios utilizaremos los tipos de datos abstractos implementados por otros desarrolladores (o por nosotros mismos), sin importarnos cómo están implementados, solo interesándonos en cómo debemos comunicarnos con dichos elementos (interfaz), confiando en que el comportamiento será el asegurado por el implementador. En este caso seríamos los conductores del automóvil.

Entonces, ¿qué es un tipo de dato abstracto?

Es un mecanismo de descripción de alto nivel que, al implementarse, genera una clase. Es decir, un TDA es un concepto matemático, mientras que una clase es la implementación del TDA. Aunque, se verá más adelante, una clase puede ser abstracta en parte o completamente. Si fuera una clase totalmente abstracta diremos que es un TDA, también. Por el momento tenemos que saber que el TDA corresponde a la etapa de diseño.

Un TDA se define indicando las siguientes cosas:

- Nombre (tipo)
- Dominio
- Operaciones
- Invariantes
- Pre y poscondiciones

El nombre nos indica al tipo que nos referimos. El dominio se relaciona con la validez de los elementos que componen el TDA, es decir, qué valores son válidos para conformar un TDA. Las operaciones surgen de la necesidad del usuario del TDA, se indican con el nombre de la operación, el dominio y el codominio. Las invariantes son, como su nombre lo indica, cosas que no varían luego de haber aplicado ciertas operaciones con determinados parámetros. Además, debemos definir pre y poscondiciones. Con algunos ejemplos ilustraremos estas definiciones.

Ejemplo 1.1. TDA Entero. Supongamos que necesitamos un tipo de dato que sea un número entero pero que no esté cubierto por los tipos básicos como *char* e *int,* podría ser porque debemos almacenar valores muy grandes que un *long int* no alcanza a representar. Entonces, generamos, en primer lugar, un TDA.

- Nombre: Entero
- Dominio: Entero ϵ Z
- Operaciones
  - $+$: Entero x Entero $\rightarrow$ Entero
  - -: Entero x Entero $\rightarrow$ Entero
  - *: Entero x Entero $\rightarrow$ Entero
  - $/$: Entero x Entero $\rightarrow$ Flotante
  - $//$: Entero x Entero $\rightarrow$ Entero

- Invariantes
  - Si tenemos un Entero e y otro x, entonces: $(e + x) - x = e$
  - Si tenemos un Entero e y otro x, entonces: $(e - x) + x = e$

El nombre tiene que ser descriptivo. Por convención, siempre comienza con mayúsculas y aplicaremos las reglas que ya hemos visto.

En dominio estamos indicando que los objetos de tipo Entero pertenecerán al conjunto infinito de los enteros. Es decir, serán un subconjunto de los enteros. Recordemos que en una máquina no podemos representar un número infinito, ya que necesitaríamos una memoria infinita.

En cuanto a Operaciones, tomemos, por ejemplo, la operación $+$, significa que, haciendo el producto cartesiano entre dos enteros (por ese motivo la x), nos devolverá otro entero. En cambio, la operación $/$ nos devolverá un Flotante. Dos comentarios con respecto a esta operación:

- Estamos asumiendo que el tipo de dato Flotante existe, también podría ser un tipo de dato básico. Debemos tener en cuenta que en la etapa de diseño no pensamos en la implementación, es decir, no estamos asociando la palabra *Flotante* con el tipo de dato *float*, por ejemplo.
- Según Meyer, esta operación debe representarse con una flecha tachada, porque no todos los enteros pertenecen al dominio de la operación. Si el segundo parámetro es un cero esta operación no se puede realizar, eso lo indicamos en la precondición. La poscondición nos dice qué resultado se obtiene siempre que las precondiciones se cumplan.

Las invariantes que señalamos, podríamos haber indicado algunas más, nos dicen que si a un Entero le sumamos otro y, a ese resultado, le restamos el mismo valor que sumamos, el Entero original no debe cambiar. Lo mismo sucede si invertimos el orden de las operaciones. Nótese que con la división entera ($//$) y la multiplicación, no hay invariante. Por ejemplo, $9 // 4 = 2$, sin embargo, $2 * 4$ no da 9.

Ejemplo 1.2. TDA Cadena. En este ejemplo diseñaremos un TDA Cadena, suponiendo que no contamos con el tipo *string*. Las pre y poscondiciones las veremos en la siguiente sección.

- Nombre: Cadena
- Dominio: $Cadena =$ "$c_1 c_2 \ldots c_n$". Los c_i son caracteres pertenecientes al conjunto Θ. $\Theta = \{A..Z\}\, U\, \{a..z\}\, U\, \{0.,9\}\, U\, \{,, -, /, (,), ", ¡, !, =, ¿, ?\}$
- Operaciones
  - cadena: $\rightarrow$ cadena
  - $+$: cadena x cadena $\rightarrow$ cadena
  - reemplazar: cadena x posición x carácter $\rightarrow$ cadena
  - longitud: cadena $\rightarrow$ entero
  - valor: cadena x posición $\rightarrow$ carácter
- Invariantes
  - valor (reemplazar (cadena, posicion, c), posicion) = c
  - longitud (cadena) = longitud (reemplazar (cadena, posicion, c))

Las operaciones las podemos clasificar en tres grupos:

a. Constructoras. Es el caso de la primera operación, la cual crea o construye una Cadena vacía. Nótese que adelante de la flecha no hay nada, es decir, no partimos de ningún elemento.

b. Modificadoras. Alteran el estado del TDA. Es el caso de las operaciones que están en segundo y tercer lugar. La operación $+$ concatena una Cadena a otra, generando una nueva Cadena. El lector se preguntará si esta operación es modificadora ya que acabamos de señalar que devuelve una nueva Cadena por lo que las originales podrían permanecer sin alterarse. Por el momento asumimos que está generando una modificación en alguno de los elementos primitivos, aunque estas decisiones las dejamos para la etapa de implementación. Con respecto a la operación reemplazar, modifica la cadena, en la posición indicada, colocando el carácter que se le pasa por parámetro.

c. Analizadoras o de consulta. No alteran el estado del TDA, sirven para consultar. Por ejemplo la operación *longitud* nos indica la cantidad de caracteres de la cadena y la operación *valor* nos devuelve el carácter que se encuentra en la posición solicitada. Ninguna de estas dos operaciones hace modificaciones.

Más adelante veremos que en C++ necesitamos, a menudo, definir operaciones destructoras. Son las encargadas de liberar recursos utilizados, generalmente es cuando debemos liberar memoria dinámica solicitada. Estas operaciones son modificadoras, ya que destruyen el TDA (cambian su estado).

En cuanto a las invariantes, decimos que si en determinada posición de una Cadena, colocamos un carácter *c*, y luego pedimos el valor que está en dicha posición, debe devolvernos el mismo carácter. En la segunda invariante aseguramos que loingitud de una Cadena no se modifica si reemplazamos un carácter por otro.

1.3. Diseño de un TDA

Como ya dijimos, en la etapa de diseño no hablamos ni pensamos en su implementación. Ni siquiera debemos tener en cuenta el lenguaje en que se implementará. El TDA debe ser independiente de su implementación. Por ejemplo, sabemos que si estamos trabajando con enteros ejecutar la operación $x + y$ debe dar el mismo resultado que realizar $y + x$, cualquiera sea su implementación y la plataforma donde se ejecute. Por este motivo, un TDA puede tener distintas implementaciones. Desarrollaremos las distintas fases (diseño, implementación y uso) con el TDA Complejo.

Ejemplo 1.3. TDA Complejo.

- Nombre: Complejo
- Dominio: Complejo $=$ (real, imaginario), con real, imaginario $\epsilon\, R$
- Operaciones
    - Complejo: $R\, x\, R \rightarrow$ Complejo

- sumar: Complejo x Complejo $\rightarrow$ Complejo
- restar: Complejo x Complejo $\rightarrow$ Complejo
- modulo: Complejo $\rightarrow R$

Para no complicar el TDA definimos solo estas cuatro operaciones. Por supuesto, las operaciones a definir dependen del uso que se pretenda dar al nuevo tipo de datos. Ahora ampliaremos los detalles de cada una de las operaciones, indicando una descripción y sus pre y poscondiciones.

Detalle de las operaciones

- Operación: Complejo
  - Descripción: construye un número complejo en base a dos parámetros, el primero conformará la parte real y el segundo la imaginaria.
  - Precondición: los dos parámetros deben ser números reales.
  - Poscondición: construye un número complejo. La parte real tendrá el valor del primer parámetro, la parte imaginaria el del segundo.
- Operación: sumar
  - Descripción: suma dos números complejos y devuelve el resultado.
  - Precondición: los dos parámetros deben ser números complejos válidos.
  - Poscondición: devuelve un número complejo tal que su parte real tendrá el valor de la suma de las partes reales de los dos parámetros y la parte imaginaria el valor de la suma de las partes imaginarias de los dos parámetros.
- Operación: restar
  - Descripción: resta dos números complejos y devuelve el resultado.
  - Precondición: condición: los dos parámetros deben ser números complejos válidos.

- Poscondición: devuelve un número complejo. La parte real tendrá el valor de la diferencia entre la parte real del primer parámetro con la parte real del segundo. La parte imaginaria será la diferencia entre la parte imaginaria del primer parámetro con la parte imaginaria del segundo.

- Operación: modulo
  - Descripción: retorna el módulo de un número complejo.
  - Precondición: el parámetro debe ser un número complejo válido.
  - Poscondición: devuelve un valor real que se calcula como la raíz cuadrada de la suma entre los cuadrados de la parte real y la parte imaginaria del parámetro.

Diseño UML

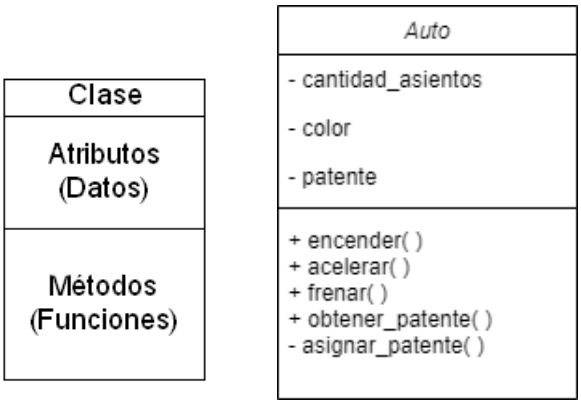
Otra forma habitual de diseñar un TDA es utilizando una simplificación de la notación UML (Lenguaje de Modelado Unificado). En UML Una clase se representa mediante un rectángulo que consiste de tres partes:

- El nombre de la clase
- Sus datos (atributos)
- Sus métodos (funciones u operaciones)

En el gráfico 1.3.1 vemos en la parte $a)$ un diseño genérico, y en la parte $b)$ un ejemplo. Los signos menos y más tienen que ver con el tipo de acceso: el menos significa privado o inaccesible del exterior y el más, público o accesible del exterior. Esto se explicará más adelante cuando veamos ocultamiento de la información.

1.4. Clases

Cuando implementamos un TDA para que pueda ser utilizado estamos creando una clase. En este momento hay que pensar en concreto: ¿qué lenguaje utili-



(a) Diseño genérico (b) Ejemplo de diseño en UML

Gráfico 1.3.1: Diseño de clase en UML

zararemos? ¿cómo haremos? ¿qué tipos de datos y estructuras usaremos dentro de la clase? Y otras preguntas más. En este momento estamos tomando el rol de implementador.

Haremos dos implementaciones distintas para la clase Complejo y las analizaremos.

En primer lugar, por norma, dividiremos el código en dos archivos distintos: el código en 1.1 es la declaración de la clase con sus atributos y métodos (operaciones). Esto va en un archivo de cabecera .h. El código que se encuentra en 1.2, archivo .cpp es en donde se definen los métodos declarados en la clase. Si bien todo puede ser escrito y definido dentro de un mismo archivo es recomendable no hacerlo, salvo como excepción en los casos en los que trabajemos con plantillas (templates) que veremos más adelante.

También, es recomendable, no definir más de una clase en un mismo archivo. Analicemos el código del archivo de declaraciones (1.1). En este archivo de extensión .h que llamaremos como el nombre del tipo de dato pero en minúscula declararemos la clase, en este caso, con la sentencia

```
class Complejo (linea 4)
```

junto con las llaves de apertura y cierre (no olvidar el punto y coma luego del cierre de llaves) definen nuestra clase Complejo. Por norma, los nombres de las clases se declaran con su primera letra en mayúsculas. En el cuerpo de la clase

irán los atributos (datos) y los métodos (funciones), pero solo la firma o cabecera de ellos. Siendo coherentes con lo que decíamos sobre la fábrica de automóviles, que no deseará que sus usuarios metan mano en el motor y que se comuniquen con el auto solo a través de su interfaz (volante, pedales, llave, etc), los atributos se colocan en una sección privada (private, en línea 6). Esto impide que el usuario pueda acceder directamente a estos atributos, por ejemplo si hubiera un objeto c de tipo $Complejo$, no podría ejecutar la siguiente sentencia:

```
c.real = 3.6;
```

Luego de los atributos se declaran las cabeceras o firmas de los métodos que en general serán públicos, ya que a través de estos métodos los usuarios interactuarán con los atributos. Si bien los nombres de los parámetros no son necesarios en esta instancia, solo su tipo, elegimos colocarlos para definir las pre y poscondiciones con los nombres de los parámetros. Por ejemplo, el constructor podría haberse declarado de la siguiente forma:

```
Complejo (double, double); (linea 16)
```

Nota. el orden de las declaraciones es indistinto. Se pueden declarar los atributos y los métodos en el orden en que uno prefiera. Incluso pueden existir varias secciones públicas y varias privadas. En general se utilizan dos estilos:

- Primero la sección privada con sus atributos y luego la parte pública con sus métodos. Este estilo decide poner en primer lugar a los atributos ya que los datos harán que, por un lado, pensemos cómo se conformarán los objetos de dicha clase. Por ejemplo, si tenemos una clase Persona en donde los atributos son nombre y telefono, el manejo de objetos de esa clase será muy distinto al de objetos de otra clase Persona con los atributos legajo, nombre y sueldo. En el primer caso podríamos pensar que los objetos servirán para una agenda, en cambio, en el segundo caso servirían para la sección de liquidación de haberes de alguna empresa.
  Por otra parte, los datos nos guían para saber qué métodos debemos implementar, aunque las operaciones estén definidas en el TDA, siempre surge alguna operación extra a la hora de implementar.

- Primero la seeción pública con sus métodos (interfaz de comunicación) y luego la parte privada con sus atributos. Este estilo interpreta que la clase la utilizará algún usuario, que es ni más ni menos que otro programador, por lo que dicho usuario solo necesita trabajar con la parte pública de la clase, no le interesa saber cómo es la parte privada según uno de los principios que mencionamos.

En la firma de los métodos debemos colocar el tipo que devuelve, el nombre del método y sus parámetros. Los constructores son métodos especiales que deben llevar el mismo nombre de la clase y no tienen tipo de retorno. Estos métodos se ejecutarán al momento de crear un objeto de la clase. Observando la firma del método $sumar$, por ejemplo, vemos que solo tiene un parámetro. Podríamos pensar que esto es un error ya que necesitamos dos números para hacer la suma. Sin embargo, el primer parámetro está dado en forma implícita y es el propio objeto que llama al método el que se pasa como parámetro. A este parámetro lo llamaremos *parámetro implícito*.

Otras sentencias que pueden llamar la atención son las dos primeras líneas:

```
#ifndef COMPLEJO_INCLUDED
#define COMPLEJO_INCLUDED
```

y la última:

```
#endif // COMPLEJO_INCLUDED
```

Estas líneas, que muchos IDEs colocan automáticamente al crear un archivo .h, sirven para no incluir o definir más de una vez la misma porción de código. Si no se recuerda esto se debe rever el primer capítulo donde se aborda el tema.

Nota. La parte pública de la clase es lo que conocemos como interfaz porque son los métodos que podrá utilizar el usuario, es decir, la comunicación con la clase. Cuando decimos *usuario* puede ser otro sistema, no necesariamente tiene que ser una persona. Por usuario se entiende "el que *usa* la clase". La firma de los métodos con sus PRE y POST condiciones debería ser lo único que el usuario tiene que saber de la clase.

En el archivo .cpp (1.2) que, por convención se llama igual que el archivo .h, salvo por su extensión, se definen todos los métodos. Nótese que hay que hacer

uso del operador de ámbito, además de incluir el archivo .h, colocando el nombre de la clase y el operador :: antes del nombre del método. De esta forma estamos indicando que el método, por ejemplo, $sumar$, es el que corresponde a la clase Complejo y no a otra clase que también podría tener un método que se llame de igual forma.

Continuando con el método $sumar$ que comienza en la línea 10, creamos un número Complejo que llamamos aux, que es donde se guardará la suma y será el objeto que devolverá. Debe observarse que, cuando hacemos referencia a real o imaginario a secas, nos estamos refiriendo a los atributos del primer parámetro que es implícito, es decir el propio objeto desde donde se llama al método. Para referirnos a los atributos del segundo parámetro, que es el único que pasamos entre paréntesis, debemos indicar el nombre del mismo, en este caso c, y con el operador "." accedemos a sus atributos.

Cabe destacar que los atributos son privados para el exterior, es decir, para afuera de la clase, pero dentro de ella podemos acceder sin restricciones.

Nota. Se incluyó el archivo $cmath$ para poder utilizar las funciones sqr (cuadrado) y $sqrt$ (raíz cuadrada).

1.5. Objetos

Decimos que existe un objeto o que creamos un objeto cuando instanciamos una clase. Es decir, cuando tenemos un elemento concreto de ese molde que llamamos clase. Por ejemplo, una clase *Auto* tendría un número de patente, una marca y un color. Un objeto de tipo *Auto* puede ser ABC 123, Renault, rojo cuyo nombre es *auto_de_mi_tio*. La relación entre un objeto y su clase es similar a la que tiene una variable con su tipo. Cuando creamos un objeto, estamos en el papel de usuario. Entonces la relación entre las etapas, los roles y los productos es la siguiente:

Etapa	Rol	Producto
Diseño	Diseñador	TDA
Implementación	Implementador	Clase
Utilización	Usuario	Objeto

Algoritmo 1.1 Archivo complejo.h - Declaración 1

```
1  #ifndef COMPLEJO_INCLUDED
2  #define COMPLEJO_INCLUDED
3
4  class Complejo {
5
6  private:
7      // atributos
8      double real;
9      double imaginario;
10
11 public:
12     // constructor con parametros
13     // PRE: re, im son de tipo double
14     // POST: construye un Complejo, en donde
15     //  real = re e imaginario = im
16     Complejo (double re, double im);
17
18     // metodo sumar
19     // PRE: c es de tipo Complejo
20     // POST: devuelve un nuevo Complejo, en donde
21     //  nuevo.real = real + c.real
22     //  nuevo.imaginario = imaginario + c.imaginario
23     Complejo sumar(Complejo c);
24
25     // metodo restar
26     // PRE: c es de tipo Complejo
27     // POST: devuelve un nuevo Complejo, en donde
28     //  nuevo.real = real - c.real
29     //  nuevo.imaginario = imaginario - c.imaginario
30     Complejo restar(Complejo c);
31
32     // metodo modulo
33     // PRE:
34     // POST: devuelve un double calculado como
35     //  raiz(real * real + imaginario * imaginario)
36     double modulo();
37 };
38
39 #endif // COMPLEJO_INCLUDED
```

Algoritmo 1.2 Archivo complejo.cpp - Implementación 1

```
1  #include "complejo.h"
2  #include <cmath>
3
4  // constructor con parametros
5  Complejo::Complejo(double re, double im) {
6      real = re;
7      imaginario = im;
8  }
9
10 // metodo sumar
11 Complejo Complejo::sumar(Complejo c) {
12     Complejo aux(0.0, 0.0);
13     aux.real = real + c.real;
14     aux.imaginario = imaginario + c.imaginario;
15     return aux;
16 }
17
18 // metodo restar
19 Complejo Complejo::restar(Complejo c) {
20     Complejo aux(0.0, 0.0);
21     aux.real = real - c.real;
22     aux.imaginario = imaginario - c.imaginario;
23     return aux;
24 }
25
26 // metodo modulo
27 double Complejo::modulo() {
28     double aux = sqr(real) + sqr(imaginario);
29     return sqrt(aux);
30 }
```

Algoritmo 1.3 Uso de la clase Complejo

```
1  #include <iostream>
2  #include "complejo.h"
3
4  using namespace std;
5
6  int main() {
7      Complejo c1(5.0, 3.0);
8      Complejo c2(4.0, 2.0);
9      Complejo c3(0.0, 0.0);
10     c3 = c1.sumar(c2);
11     cout << "el modulo de c3 es " << c3.modulo() << endl;
12     return 0;
13 }
```

¿Cómo se crea un objeto?

Similar a una variable: se indica el nombre de la clase (tipo), un nombre para el objeto y si el constructor lleva parámetros se deben pasar entre paréntesis. En las líneas 7, 8 y 9 del algoritmo 1.3 creamos tres objetos de tipo *Complejo*. En la línea 10 llamamos al método *sumar* a través del objeto $c1$. Este es el primer parámetro que mencionábamos. El segundo parámetro es $c2$. El método *sumar* devuelve un objeto de tipo *Complejo* que lo asigna a $c3$.

Otro implementador podría haber pensado una estructura distinta para la clase *Complejo*. Por ejemplo, en lugar de utilizar dos atributos: uno para la parte real y otro para la imaginaria, podría pensar en utilizar un vector. Con esta segunda implementación tenemos un vector de tipo double, este vector, de tamaño dos, lo llamamos z. En la primera celda guardaremos la parte real y en la segunda,la parte imaginaria. Lo interesante es notar que (ver 1.4), del archivo .h sólo cambia la parte privada y, por supuesto, cambia la definición de los métodos en el archivo .cpp (ver 1.5). Sin embargo, para el usuario estos cambios son transparentes porque la interfaz no cambió en nada, por lo que el programa que utiliza el tipo de dato *Complejo* funciona a la perfección con ambas implementaciones. Esta es la ventaja de separar la interfaz de la implementación o, pensado de otra forma, la ventaja de *abstraerse*: no pensar cómo están hechas las cosas, sino cómo debemos tratar con ellas.

¿Por qué esto es importante? ¿Qué necesidad puede haber para cambiar una

Algoritmo 1.4 Archivo complejo.h - Declaración 2

```
1  #ifndef COMPLEJO_INCLUDED
2  #define COMPLEJO_INCLUDED
3
4  class Complejo {
5  private:
6      // atributos
7      double z[2];
8
9  public:
10     // constructor con parametros
11     // PRE: re, im son de tipo double
12     // POST: construye un Complejo, en donde
13     //     real = re e imaginario = im
14     Complejo (double re, double im);
15
16     // metodo sumar
17     // PRE: c es de tipo Complejo
18     // POST: devuelve un nuevo Complejo, en donde
19     //     nuevo.real = real + c.real
20     //     nuevo.imaginario = imaginario + c.imaginario
21     Complejo sumar(Complejo c);
22
23     ...
24 };
25
26 #endif // COMPLEJO_INCLUDED
```

implementación? La implementación se puede querer cambiar por una cuestión de eficiencia, velocidad en la ejecución, una mejora en el mantenimiento de la clase, facilidad para agregar funcionalidad, etc. Estamos acostumbrados a que salgan nuevas versiones de las aplicaciones que utilizamos, con una interfaz más atractiva, amigable y nueva funcionalidad (algunas veces, esto no siempre sucede). Sin embargo, las nuevas versiones en general son capaces de trabajar con los archivos que hemos creado con versiones anteriores. De lo contrario tendríamos que generar un nuevo archivo para cada nueva versión que se lance. Ese es uno de los objetivos a perseguir cuando desarrollamos clases y trabajamos con objetos.

Algoritmo 1.5 Archivo complejo.cpp - Implementación 2

```
#include "complejo"
#include <cmath>

// constructor con parametros
Complejo::Complejo(double re, double im) {
    z[0] = re;
    z[1] = im;
}

// metodo sumar
Complejo Complejo::sumar(Complejo c) {
    Complejo aux(0.0, 0.0);
    aux.z[0] = z[0] + c.z[0];
    aux.z[1] = z[1] + c.z[1];
    return aux;
}
...
```

1.6. Algunas características de la POO

1.6.1. Encapsulamiento

El término encapsulamiento se refiere a la idea de "encapsular" los datos que distinguen a una clase junto con las operaciones que tratan a los mismos. Hablando de manera informal, metemos en una bolsa (cápsula) todo junto: los datos y las operaciones que se necesiten para trabajar con ellas como una unidad. A diferencia de la programación estructurada en donde teníamos datos por un lado y funciones que operaban con ellos de forma separada.

1.6.2. Ocultamiento de la información

Con la finalidad de proteger al objeto y a su usuario, los datos deberían ser, en general, inaccesibles desde el exterior para que no puedan ser modificados sin autorización. Por ejemplo: sería un caos si, en una biblioteca, cada persona tomara y devolviera por su cuenta los libros que necesitara ya que podría guardarlos en otro lugar, mezclarlos, dejar a otros usuarios sin libros, etc. Para que no suceda esto hay una persona, que es el bibliotecario, que funciona de interfaz entre el lector y los libros. El lector solicita al bibliotecario un libro. El bibliotecario

toma del estante correspondiente el libro, toma la credencial de la persona que lo solicitó, lo registra y se lo presta.

Este es uno de los principios que se utilizan en la POO: colocar los datos de manera inaccesibles desde el exterior y poder trabajar con ellos mediante métodos estipulados a tal fin. Estos métodos forman parte de la interfaz del usuario.

1.6.3. Herencia

Tenemos en claro que un objeto *auto* no es lo mismo que un objeto *ómnibus* o *ferrocarril*. Pero, todos estos objetos tienen cosas en común. Lo más importante que comparten es que todos sirven como medio de transporte (ver gráfico 1.6.1). En este punto aparece un nuevo concepto en la POO que es la herencia. Las clases Auto, Colectivo, Ferrocarril heredan de una clase en común que será Transporte. Por lo tanto, tendrán comportamientos (métodos) y atributos que compartirán. Como ser la cantidad de personas que pueden transportar, el consumo de combustible, etc. Obviamente, luego, cada uno de los objetos mencionados podrá tener otros atributos y métodos que serán particulares de su clase.

Hay que notar que a las clases Auto, Colectivo y Ferrocarril pudimos ilustrarlas con una fotografía. Sin embargo, la clase Transporte, no. En estos casos, decimos que la clase Transporte es una clase abstracta. Una clase abstracta es aquella que no se puede instanciar, es decir, no se puede crear un objeto de esa clase. En este ejemplo no tendría sentido crear un objeto de tipo Transporte ya que no sería real. Decimos que las clases concretas o efectivas son las que se pueden instanciar, en este ejemplo, las que heredan de Transporte.

Nota. La herencia no siempre implica clases abstractas. Por ejemplo, una clase *Persona* podría ser una clase real (o concreta) y, la clase *Estudiante*, heredaría de persona. Es decir, sería una *Persona* con algunas características más.

Un diagrama UML podría ser el que se muestra en el gráfico 1.6.2. Vemos que la clase Auto tiene un solo dato, que es espacio_en_el_baul, sin embargo, esto no es cierto, ya que heredó de la clase Transporte cantidad_asientos, color

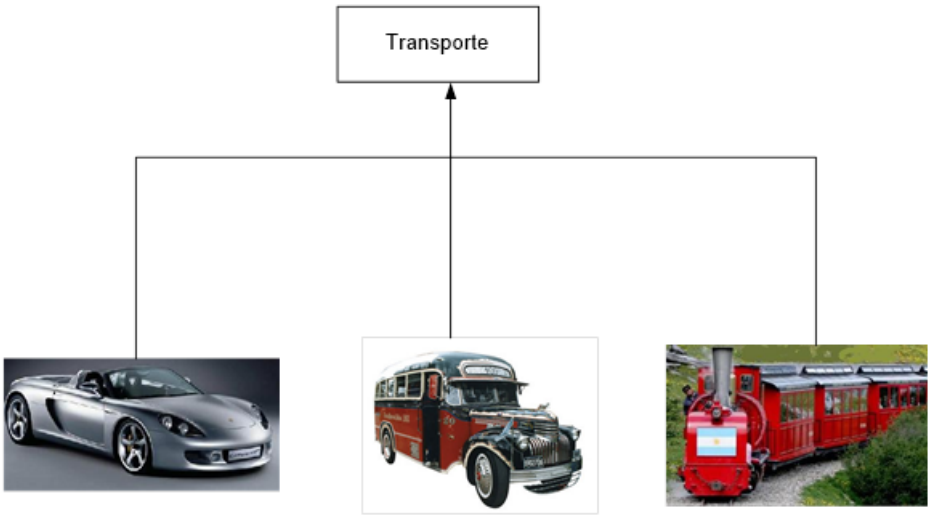


Gráfico 1.6.1: Herencia

y patente. Lo mismo sucede con sus métodos.

1.6.4. Polimorfismo

El polimorfismo significa que un objeto puede tener varias (poli) formas (morfo). Por un lado, podemos pensarlo como la propiedad que tienen distintos objetos de comportarse de distinta manera ante un mismo mensaje. Por otro lado, podríamos verlo desde el punto de vista de las estructuras. Una estructura polimorfa es la que puede albergar objetos de distintas clases. Por ejemplo, un vector almacena figuras geométricas: triángulos, cuadrados, rectángulos y círculos. Son diferentes objetos, con características distintas que conviven en el mismo array.

Si bien veremos este tema en el próximo capítulo, con respecto al comportamiento de los objetos ante un mismo mensaje, anticipamos el concepto con un ejemplo: si a un objeto *Persona* le decimos que se alimente, no actuará de la misma forma que si se lo dijéramos al objeto *Planta*. Entonces, el mismo método: *alimentar* trabajaría de forma diferente dependiendo qué objeto lo invoque. Esto puede no ser novedoso si supiéramos cuál es ese objeto, la gracia del polimorfismo es no saber qué objeto lo está invocando.

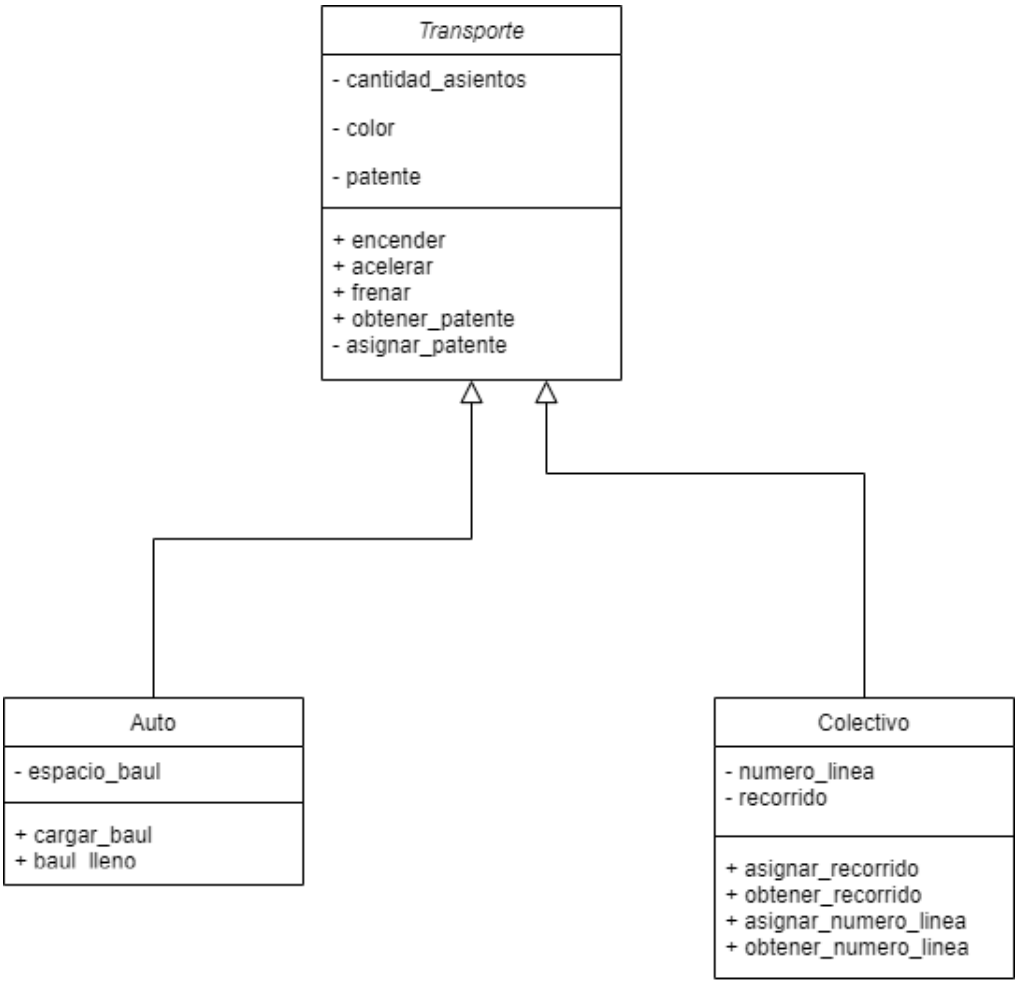


Gráfico 1.6.2: Herencia: diagrama UML

1.7. Acceso a los atributos

Unas secciones atrás recomendamos que los atributos deben ser privados con el fin de proteger los datos. En la clase *Complejo*, tanto el atributo *real* como *imaginario* son de tipo privado. Por este motivo debemos definir métodos para poder asignar valores a estos atributos. Hay que tener en cuenta que no nos sirve un solo método para asignar ambos valores, porque podríamos desear asignar solo la parte real y no modificar la parte imaginaria o viceversa. Entonces, debemos hacer un método para asignar su parte real y, otro, para su parte imaginaria.

En general, como regla, por cada atributo, tendremos un método específico para asignar o colocar su valor. Estos métodos, los llamamos asignar_nombre_atributo o con algún sinónimo de *asignar*. Por ejemplo asignar_real o asignar_imaginario.

El caso contrario sucede cuando tenemos la necesidad de consultar esos valores o mostrarlos. Tampoco podríamos acceder a dichos atributos, por lo que deberíamos tener los métodos para conseguir dichos valores, estos métodos se llaman por convención obtener_nombre_atributo. Por ejemplo obtener_real u obtener_imaginario.

Nota. Podrían llamarse de cualquier manera, se destaca que es solo por convención. Otros nombres utilizados en lugar de *asignar* son *cambiar* o *modificar*.

Obviamente, los métodos de *asignar* y *obtener* tienen que ser públicos para poder ser accedidos fuera de la clase.

Los métodos *asignar* colocan el valor del atributo, el cual se deberá pasar por parámetro y no devuelven nada. En cambio los métodos *obtener* no deben recibir ningún parámetro pero sí deberán devolver un valor que es el del atributo. Por ejemplo con la clase Complejo, los métodos asignar son:

```
void Complejo::asignar_real(double r) {
    real = r;
}

void Complejo::asignar_imaginario(double i) {
    imaginario = i;
}
```

Y los de obtener:

```
double Complejo::obtener_real() {
    return real;
}

double Complejo::obtener_imaginario() {
    return imaginario;
}
```

1.8. Constructores

Habíamos dicho que los constructores son métodos especiales que no tienen ningún tipo de retorno, ni siquiera *void*, y que se llaman de forma automática cuando creamos un objeto. Estos métodos son tan importantes que, en versiones anteriores de C++, si no escribíamos uno, el mismo compilador generaba uno de oficio, cuya única tarea era inicializar todos los atributos numéricos en cero, los booleanos en *false* y los punteros en *null*. Para el desarrollador era transparente ya que el código no estaba escrito pero se llamaba al momento de crear un objeto.

Sin embargo, en las nuevas versiones, si no inicializamos a los atributos con algún valor específico, contendrán lo que conocemos como basura. Esta modalidad va en línea con el lenguaje Java.

Es decir, si no hubiéramos escrito un constructor para la clase Complejo deberíamos haber inicializado las variables de alguna de estas dos maneras:

```
double real = 0.0;
double imaginario = 0.0;
```

o

```
double real {0.0};
double imaginario {0.0};
```

Como veremos en la siguiente sección, podemos tener más de un constructor.

1.9. Sobrecarga de métodos

Dos o más métodos pueden tener el mismo nombre siempre y cuando difieran en

- la cantidad de parámetros
- los tipos de los parámetros
- ambas cosas

La sobrecarga vale también para los constructores. Imaginemos que quisiéramos construir un objeto de tipo *Complejo* sin asignarle en principio ningún valor en particular, que por defecto se inicialicen en 0 tanto la parte real como la imaginaria. Como el constructor lleva dos parámetros la siguiente línea

```
Complejo c;
```

da error en tiempo de compilación. Por lo que estamos obligados a escribir

```
Complejo c(0.0, 0.0);
```

Sin embargo, una solución es sobrecargar el constructor, es decir, escribir otro constructor que no lleve parámetros, de la siguiente forma:

```
// Constructor sin parametros - definicion
Complejo::Complejo() {
    real = 0.0;
    imaginario = 0.0;
}
```

Entonces, si luego, como usuarios escribiéramos

```
Complejo c1;
Complejo c2(3.5, 2.4);
```

se ejecutaría el constructor sin parámetros para el objeto $c1$, asignando el valor de cero para la parte real y la imaginaria. En cambio, para la creación del objeto $c2$ llamaría al constructor con parámetros.

Nota. Siempre es recomendable escribir un constructor por defecto (sin parámetros), en especial si pudieran crearse vectores de objetos de la misma clase, ya que no se podrían pasar los n parámetros de manera explícita.

En el caso de escribir un constructor que deje los valores de inicialización por defecto, el código sería el siguiente (en el mismo archivo .h):

```
Complejo( ) = { };
```

o

```
Complejo( ) = default;
```

1.10. Parámetros por defecto

El ejemplo anterior de sobrecarga de constructores también se puede solucionar con lo que se llama parámetros por defecto. ¿Qué es y cómo funciona? Lo vemos con un ejemplo:

```
// Constructor con parametros por defecto
Complejo::Complejo(double re = 0.0, double im = 0.0) {
    real = re;
    imaginario = im;
}
```

Un constructor como el anterior nos sirve para las siguientes sentencias:

```
Complejo c1;
Complejo c2(3.5, 2.4);
Complejo c3(3.5);
```

En el caso de tener dos parámetros, como en la creación del objeto $c2$, se ejecuta el cuerpo del constructor colocando el valor de 3.5 para la parte real y 2.4 para la imaginaria. En caso de no tener parámetros, como en la creación del objeto $c1$, utiliza los valores por defecto que, en este ejemplo, valen 0.0 para sus dos atributos. En cuanto a la creación del objeto $c3$, el valor 3.5 es asignado a su parte real porque es el primer atributo y, para su parte imaginaria, como falta el valor, toma el valor por defecto que es 0.0.

Los parámetros por defecto no pueden solucionar el caso inverso de la construcción del objeto $c3$: si quisiéramos que tome por defecto a la parte real y que asigne el valor pasado por parámetro al atributo *imaginario*, deberíamos indefectiblemente pasar los dos parámetros.

1.11. El puntero *this*

Todo objeto se crea con una referencia o puntero a sí mismo, este puntero se llama *this* y se genera de manera automática. ¿Cuándo es necesario utilizarlo? Se puede utilizar siempre pero nos vemos obligados a usarlo cuando el compilador no pueda resolver una ambigüedad en los nombres. Veamos algunos ejemplos:

```
// No se usa this
Complejo::Complejo(double re, double im) {
```

```
    real = re;
    imaginario = im;
}

// Se utiliza this pero no es necesario
Complejo::Complejo(double re, double im) {
    this->real = re;
    this->imaginario = im;
}

// Es necesario utilizar this
Complejo::Complejo(double real, double imaginario) {
    this->real = real;
    this->imaginario = imaginario;
}
```

En el último caso, es necesario utilizar el puntero *this* para diferenciar los parámetros *real* e *imaginario* de los atributos del propio objeto (this).

1.12. Atributos y métodos estáticos

Un atributo (o campo) estático es un atributo que no depende de un objeto en particular sino de la clase en sí. Por ejemplo, un objeto de la clase *Auto* tendrá un atributo *color*. Podríamos tener varios objetos de tipo *Auto* que tuvieran el mismo color. Sin embargo, el dato o atributo *patente* no puede repetirse. Por lo que ese atributo no debería manipularse ni cambiarse desde afuera. Además, debe tener valores correlativos que, de alguna forma debería consultarse cuál fue la última patente otorgada en el momento de establecerla o, cuál es la próxima patente a asignar. Este atributo, *patente_a_asignar*, no debería pertenecer a un objeto *Auto* en particular, sino a la clase *Auto* para poder controlar sus valores desde allí. Esto se logra indicando que el atributo *patente_a_asignar* es *static*. Cabe resaltar que por clase existe una sola copia de un campo que es *static*.

En el gráfico 1.12.1 representamos el ejemplo anterior. Vemos que cada auto tiene un *color* y una *patente*. Pero esa patente no puede ser cualquiera, conserva un orden, por lo tanto, el valor del atributo *patente* se controla por el atributo *patente_a_asignar* que es único para toda la clase.

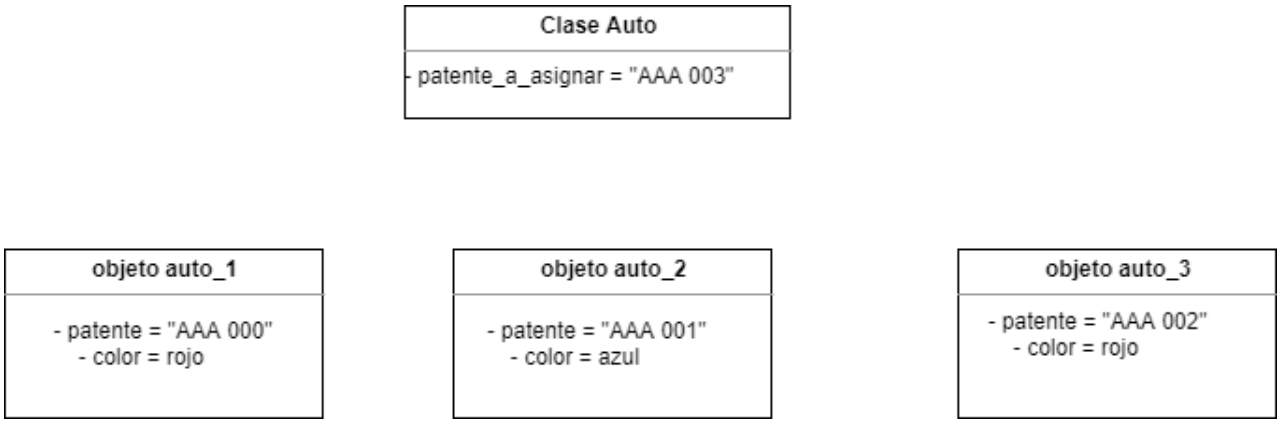


Gráfico 1.12.1: campos estáticos

Para que un atributo sea *static*, solo se debe agregar este modificador en el momento de declararlo. Por otro lado, los campos estáticos pueden ser accedidos desde cualquier objeto, por ejemplo:

```
auto_1.patente_a_asignar
```

Sin embargo, por una cuestión de claridad en el código, es preferible accederlos desde la misma clase:

```
Auto::patente_a_asignar
```

De todas formas, siendo coherentes con lo que venimos diciendo, el atributo *patente_a_asignar*, debería ser privado para no ser accedido desde fuera de la clase. Entonces, la forma más lógica de acceder a un atributo privado es mediante algún método escrito para tal fin. Sin embargo, como este atributo además de privado es estático, debería accederse mediante un método estático.

Nota. Los atributos estáticos deben ser inicializados ya que su valor debe estar disponible para la clase y no para un objeto en particular.

En el algoritmo 1.6 se muestra un ejemplo de cómo se definen e inicializan los atributos estáticos. Analizamos ese código: definimos una clase con nombre *ClaseA*, en un archivo .h, como siempre. Esta clase tiene un atributo x, que pertenece a cada objeto en particular. Pero, además, tiene un atributo y que pertenece directamente a la clase porque es estático.

Supongamos que deseamos que los objetos se vayan creando con un número

Algoritmo 1.6 Atributos y métodos estáticos

```
// Archivo claseA.h
class ClaseA {
private:
    int x;      // Atributo x (propio del objeto)
    static int y;   // Atributo y estático (propio de la clase)
public:
    ClaseA(int a); // Constructor
    // Metodos obtener (el de y debe ser estático)
    int obtener_x();
    static int obtener_y();
};

// Archivo claseA.cpp
#include "claseA"

int ClaseA::y = 100; // El atributo y se debe inicializar

ClaseA::ClaseA(int x) {
    this->x = x;
    y++;
}

int ClaseA::obtener_x() {
    return x;
}

int ClaseA::obtener_y() {
    return y;
}
```

de serie que empieza en 101, sigue en 102, etc. Debemos inicializar el atributo y, como hacemos en el archivo claseA.cpp. Cabe destacar que esta inicialización está fuera de cualquier método.

Por otra parte, en el constructor nos encargamos de incrementar en uno a esta variable. Es decir, cada vez que creamos un objeto de tipo *ClaseA*, incrementamos este contador.

En la función main (ver 1.7), creamos dos objetos de tipo *ClaseA* pasando los parámetros para que inicialice el valor de x pero no para y, ya que este valor depende de la clase y no de cada objeto en particular.

El llamado para imprimir los valores de y se puede realizar tanto desde un

Algoritmo 1.7 Uso de atributos estáticos

```
#include <iostream>
#include "claseA"

using namespace std;

int main() {
    ClaseA a(3);
    cout << "Objeto a, atributo x: " << a.obtener_x() << endl;
    // Se llama al metodo obtener_y desde el objeto
    cout << "ClaseA, atributo y: " << a.obtener_y() << endl;
    ClaseA b(5);
    cout << "Objeto b, atributo x: " << b.obtener_x() << endl;
    // Se llama al metodo obtener_y desde la clase
    cout << "ClaseA, atributo y: " << ClaseA::obtener_y() << endl;
    return 0;
}
```

objeto, como hacemos en el primer caso, como desde su clase, como vemos en la última impresión de línea. Es preferible hacerlo de esta forma dado que en el primer caso pareciera que el atributo es un valor perteneciente al objeto y no a su clase.

La salida de la ejecución del código 1.7 es:

```
Objeto a, atributo x: 3
ClaseA, atributo y: 101
Objeto b, atributo x: 5
ClaseA, atributo y: 102
```

1.13. Destructores

Los destructores, al igual que los constructores, son métodos especiales que se ejecutan sin un llamado explícito. La función que tienen es liberar recursos, como cerrar archivos, liberar memoria dinámica utilizada, etc. En el ejemplo de la clase *Complejo* que estuvimos viendo no era necesario liberar ningún recurso, por este motivo no programamos ningún destructor.

Como sucede con los constructores, los destructores deben tener el mismo nombre que la clase, pero con un signo ~ adelante. Además, no llevan parámetros ni pueden ser sobrecargados. Una sobrecarga no sería permitida por el lenguaje ya

que se estaría repitiendo la firma del método, además de que no tendría ningún sentido.

Es importante recalcar que los destructores no deben llamarse en forma explícita, se llaman automáticamente cuando:

- En caso de un objeto no dinámico, se termina el ámbito en donde el objeto fue definido
- Si el objeto es dinámico, cuando se ejecuta la instrucción *delete*. Esto sucede cuando el objeto fue creado con el operador new.

Por ejemplo:

```
class Clase {
    // atributos ...
    // metodos ...
    // constructores
    Clase();
    ...
    // destructor
    ~Clase();
};

void funcion() {
    Clase o1; // Se crea un objeto no dinamico de tipo Clase
    Clase *ptr = new Clase(); // Se crea un objeto dinamico tipo Clase
    ...
    delete ptr; // Se libera la memoria solicitada
                // En este momento se llama al destructor de Clase
    ...
    // Otras sentencias
    ...
    // Termina el bloque donde se define funcion
} // En este momento se llama al destructor del objeto o1
```

¿Cuándo y cómo programamos un destructor?

El caso típico es cuando se solicita memoria dinámica para uno o más atributos de la clase. Lo veremos con otro ejemplo. Supongamos que deseamos

una clase para manejar vectores de enteros en forma dinámica, indicando, en su constructor el tamaño del vector. Entonces:

```
class Vector {
private:
    // atributos
    int tamanio; // longitud del vector
    int *datos; // datos que tendra en el vector

public:
    // constructor
    Vector(int tam);

    // otros metodos
    ...
    // destructor
    ~Vector();
};

// Definicion de los metodos
// Constructor
Vector::Vector(int tam) {
    tamanio = tam;
    // Se solicita memoria dinamica para guardar los datos
    datos = new int[tamanio];
}

// Destructor
Vector::~Vector() {
    // Se libera la memoria solicitada
    if (tamanio > 0)
        delete []datos;
}
```

1.14. Constructor de copia

El constructor de copia no es ni más ni menos que otro constructor más, el cual, si no es programado, el lenguaje provee uno de oficio.

¿Cuándo se llama? Se llama cuando se necesita copiar un objeto, por ejemplo cuando se pasa un objeto por parámetro en alguna función o, como cuando se crea un objeto igualándolo a otro. Ejemplos:

```
1 void funcion(Clase o) {
2     // sentencias de la funcion
3     ...
4 }
5
6 Clase o1;
7 funcion(o1);
8 Clase o2 = o1;
```

En la línea 7 del código anterior se llama a la función definida al principio, la cual recibe un objeto por parámetro. Dentro de la función se debe trabajar con este objeto el cual está pasado por valor, es decir que se crea, internamente, una copia del mismo, llamando al constructor de copia.

En la última línea (línea 8), se crea un objeto $o2$ igualándolo a otro objeto $o1$ de su misma clase. También se crea una copia de este objeto, ya que se necesita construir el objeto $o2$ en base al objeto $o1$.

¿Qué hace el constructor de copia de oficio? Simplemente copia todos los valores de los atributos de un objeto al otro.

¿Qué problema puede presentar esto? El problema surge cuando se utiliza memoria dinámica. Volvamos al ejemplo de la clase *Vector*. Esta clase *Vector* tiene dos atributos: un entero que indica su tamaño y un puntero, a una zona de memoria donde estarán los datos en sí. Supongamos que creamos un objeto de tipo *Vector* para guardar los datos 1, 2 y 3. Como deseamos guardar tres enteros, creamos un objeto Vector con parámetro 3 de la siguiente manera:

```
Vector vec1(3);
```

Luego, con otro método colocaremos los datos 1, 2 y 3 en el objeto. Pero analicemos un poco la línea de código anterior, luego de ejecutarse dicha línea, se llamará al constructor que hará lo siguiente:

- el atributo *tamanio* quedará con el valor 3

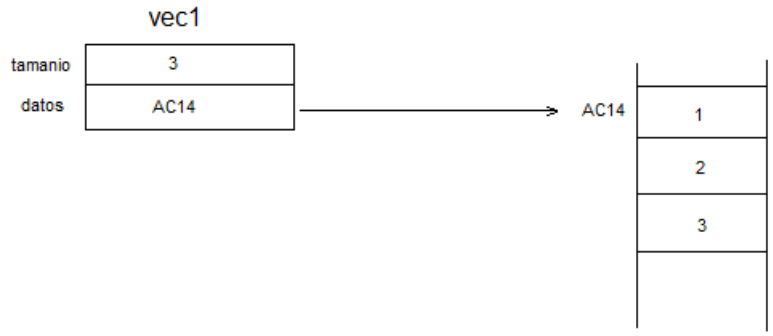


Gráfico 1.14.1: Atributos que manejan memoria dinámica

- luego, pedirá memoria para albergar a tres enteros. La porción de memoria que asigne tendrá una dirección de comienzo, supongamos que esta dirección es AC14. Por lo tanto, este valor, AC14, será el dato que guarde el atributo *datos*.

Hay que tener muy en claro que, los valores 1, 2 y 3, que aún no se han guardado pero se supone que se harán a la brevedad con otro método, no están almacenados dentro del objeto *vec1*, sino que están fuera de él, en otra porción de memoria. En particular, ocuparán tres lugares contiguos para que cada uno albergue un entero a partir de la dirección AC14. Lo único que tenemos en el objeto *vec1* que se relaciona con estos datos es dicha dirección de memoria. La representación gráfica la vemos en 1.14.1.

Ahora bien, cuando otro objeto, supongamos *vec2* copie de *vec1*, copiará todos los valores de los atributos tal cual están. El constructor de copia toma lo que tiene el atributo datos como un entero más, no sabe si eso es una dirección u otra cosa. Por lo tanto, si ejecutamos la siguiente sentencia:

```
Vector vec2 = vec1;
```

Nuestro nuevo objeto, vec2, tendrá el valor de 3 en *tamanio*, lo cual es correcto y, en *datos*, tendrá AC14, es decir, apuntará a la misma zona de memoria que *vec1*. Esto no está bien por varios motivos: uno de los problemas es que, si modificamos los valores de *vec2*, también se modificarán los de *vec1* y viceversa. En general esto no es lo que uno desea. Pero hay otro motivo por lo que lo anterior no es correcto, y este motivo hará que nuestra aplicación finalice de manera incorrecta: cuando el objeto *vec1* (o *vec2*) sea destruido, porque se terminó el

ámbito de su definición, por ejemplo, se llamará al destructor y se liberará la memoria solicitada, lo que hará inaccesible los datos para el otro objeto, ya que la memoria se liberó. Es decir que un intento de acceso a la porción de memoria a través del otro objeto daría un error en tiempo de ejecución. También daría un error en tiempo de ejecución cuando se llame al destructor del objeto que siguió con vida, ya que se intentaría liberar memoria que ya ha sido liberada.

En conclusión: siempre que manejemos atributos que utilicen memoria dinámica es necesario definir un constructor de copia.

El parámetro del constructor de copia debe ser una referencia a un objeto de la misma clase que estamos programando, dado que se va a copiar de él. No puede ser el objeto en sí, es decir, no se podría pasar por valor, porque se debería, a su vez, copiar el mismo parámetro, lo que crearía un llamado recursivo infinito imposible de resolver. Además, por convención, se suele indicar que esa referencia es constante para asegurar que el objeto no será modificado. Entonces, siguiendo con el ejemplo de la clase *Vector*:

```
// Constructor de copia
Vector::Vector(const Vector &vec) {
    // los tamanios deben ser iguales
    tamanio = vec.tamanio;
    // Se solicita nueva memoria para guardar los datos
    datos = new int[tamanio];
    // Se copian los valores a esa nueva porcion de memoria
    for (int i = 0; i < tamanio; i++)
        datos[i] = vec.datos[i];
}
```

De esta forma, al crear una copia del objeto *vec1*, se estaría creando otra copia de los datos en otra porción de memoria. Con lo cual, cada objeto tendría la dirección de distintas zonas de memoria, siendo independientes entre sí.

Nota. El llamado a los distintos constructores se realiza muchas veces sin que nos demos cuenta. Como ejercicio es aconsejable escribir un cartel de aviso dentro de cada constructor diciendo "Se llama al constructor de..." para verificar los llamados a los diferentes constructores.

1.15. Sobrecarga de operadores

Al igual que con los métodos, los operadores, también pueden sobrecargarse. Si se tiene que sumar dos números, lo más lógico es desear utilizar el operador + y no un método que se llame sumar, aunque estos números fueran números complejos o de cualquier otro tipo. De hecho, el operador + lo venimos utilizando con sobrecarga, por ejemplo:

```
int x = 5, y = 8, z;
string s1 = "Hola ", s2 = "Juan", s3;
z = x + y;
s3 = s1 + s2;
```

La variable z queda con el valor 13 y la variable s3 con el string "Hola Juan". Lo que significa que si los parámetros son enteros el operador + realiza la suma algebraica. En cambio, si los parámetros son de tipo string, el operador + los concatena.

Para sobrecargar el operador + en la clase *Complejo*, debemos agregar en el archivo .h, el siguiente método:

```
// operador +
Complejo operator+ (Complejo c);
```

Lleva las mismas PRE y POST condiciones que el método sumar. Luego, en el archivo .cpp definimos:

```
// operator+
Complejo Complejo::operator+(Complejo c) {
    Complejo aux;
    aux.real = real + c.real;
    aux.imaginario = imaginario + c.imaginario;
    return aux;
}
```

Como se observa, la sintaxis es exactamente la misma que la del método sumar.

¿Cómo es su uso? La primera forma de uso sería la misma que la de cualquier otro método, por lo tanto, podríamos escribir la siguiente sentencia:

```
c3 = c1.operator+(c2);
```

Sin embargo, la sobrecarga de operadores tiene sentido con la siguiente forma de uso:

```
c3 = c1 + c2;
```

Como se advierte, estamos realizando la suma de números complejos con la misma sentencia que si hiciéramos la suma de enteros o flotantes.

Sobrecarga del operador =

Los problemas indicados con el constructor de copia, se repiten cuando se utiliza el operador $=$. Por ejemplo:

```
vec2 = vec1;
```

En este caso no se utiliza el constructor de copia, porque los constructores son invocados cuando el objeto se crea, sin embargo, en la sentencia anterior, no se están creando objetos, se supone que ya están creados anteriormente. De esta forma se está utilizando una sobrecarga del operador $=$. Por lo tanto, deberíamos también sobrecargar el operador $=$ para poder utilizarlo sin que se produzcan efectos no deseados.

Hay que tener cuidado distinguiendo estas sentencias:

```
Vector vec2 = vec1; // llama al constructor de copia
```

En la sentencia anterior no llama al operador $=$ porque se está creando el objeto en ese momento, es equivalente a escribir:

```
Vector vec2(vec1);
```

El código del operador $=$ es similar al del constructor de copia, con la diferencia de que hay que tener presente que el objeto ya está creado, por lo que debe verificar si tiene memoria solicitada anteriormente y liberarla. En el siguiente capítulo vamos a desarrollar por completo la clase *Vector* y veremos bien este tema.

1.16. Pre y pos condiciones

Es importante definir al principio de cada método la Pre y Post condición. ¿Qué significa esto? ¿Por qué son necesarias?

Las Pre y Post condiciones son comentarios que, además de aclarar el código del método, funcionan como un contrato entre el usuario y el implementador. En una precondición decimos en qué estado debe estar el objeto para llamar a determinado método y cuáles son los valores válidos de sus parámetros. En la poscondición decimos cómo va a quedar el estado de la clase luego de ejecutar el método y qué devuelve, si hubiera alguna devolución.

Estos comentarios son necesarios por las siguientes dos razones:

a. La primera razón es porque ayuda a que el código sea más claro. A la hora de extenderlo o buscar errores de ejecución u otros motivos, estos comentarios nos ayudan a ver qué hace esa porción de código.

b. La segunda razón es que forman parte del contrato implementador – usuario. El implementador o desarrollador de la clase dice lo siguiente: "si se cumplen las precondiciones, garantizo el resultado del método". De esta manera, son un buen elemento para determinar responsabilidades. ¿Por qué falló el sistema? ¿Se cumplió la precondición? Si la respuesta es sí, el sistema falló por un desacierto del implementador. En cambio, si la respuesta es no, el responsable de la falla es el usuario de la clase.

1.17. Una clase compuesta

Veamos una clase más compleja que la clase Complejo, valga el juego de palabras. Supongamos que queremos implementar la clase *Novela*, la cual tiene un título, una cantidad de palabras y un *Autor*. Lo vemos en el gráfico 1.17.1 UML simplificado. Una Novela tiene, como atributos, un título, un año en la que fue escrita y un Autor. Pero, a su vez, el Autor, será un objeto que tiene un nombre, un lugar de nacimiento y un lugar de residencia.

Nota: elegimos estos atributos para señalar algunas cuestiones interesantes pero, obviamente, en una aplicación real los atributos podrían y deberían ser

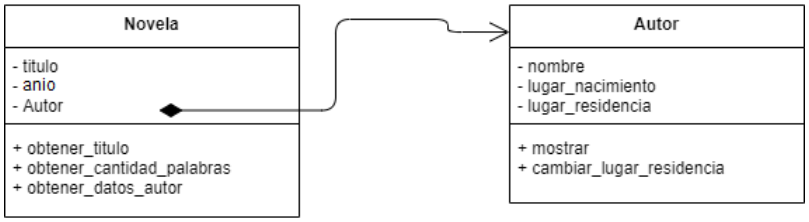


Gráfico 1.17.1: Composición

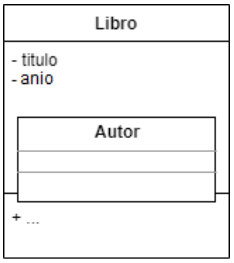


Gráfico 1.17.2: Composición explícita

muchos más.

Si pensamos que un objeto de tipo Autor está dentro de la clase Libro, deberíamos hacer un diagrama como el que vemos en la figura 1.17.2. Sin embargo, UML tiene diferentes tipos de conectores para marcar esta diferencia, ya veremos algunos pero, por el momento, no nos preocuparemos por los diagramas. Veamos cómo queda la implementación y cómo sería el uso de estas clases.

Comenzamos por la clase más simple: Autor. El código lo vemos en los algoritmos 1.8 y 1.9. Tiene tres atributos de tipo string con algunos métodos. Nota: definimos la cantidad mínima de métodos para no distraer del objetivo central en donde queremos poner el foco. También faltaría el diseño y las pre y poscondiciones.

Si bien el destructor no se necesita vemos, en la definición de este método y en la definición de los constructores, que hay unos carteles comentados indicando qué constructor se utiliza en cada momento y cuándo se ejecuta el destructor. Esta práctica es una buena idea solo en la etapa de aprendizaje, para observar cuándo se crean los objetos y cuántos. En general observamos más creaciones de objetos de la que esperaríamos.

Asumimos que los métodos de esta clase no presentan ninguna novedad por

Algoritmo 1.8 archivo autor.h

```
#ifndef Autor_H_INCLUDED
#define Autor_H_INCLUDED
#include <string>

class Autor {
private:
    // Atributos
    std::string nombre;
    std::string lugar_nacimiento;
    std::string lugar_residencia;

public:
    // Metodos
    Autor(std::string n, std::string ln, std::string lr);
    Autor(std::string n);
    Autor(Autor& p);
    ~Autor();

    void mostrar();
    void cambiar_lugar_residencia(std::string lr);
};
#endif // Autor_H_INCLUDED
```

lo que no los comentamos.

Luego, en los algoritmos 1.10 y 1.11, tenemos el código para la implantación de la clase Novela. Esta clase tiene un título, el año de la novela y su autor. También declaramos más de un constructor, el destructor (que podíamos haber obviado) y un par de métodos más.

La definición de estos métodos necesita algunas aclaraciones. Como Novela, tiene un atributo de tipo Autor, necesita crear este objeto antes que nada. Tengamos en cuenta que Autor no tenía un constructor por defecto, es decir, sin parámetros, por lo que necesitamos crear el objeto pasándole alguno de los parámetros aceptados. El constructor que comienza en la línea 4 espera un objeto *Autor* ya creado, por lo que usará el constructor de copia. Ese objeto debe crearse antes de acceder al código del método. ¿Cómo se hace eso? Se utiliza lo que llamamos inicializadores: luego de los parámetros del método escribimos dos puntos (:) y el nombre del atributo / objeto *autor*, con el parámetro actual. En este ejemplo es *a*. Luego, en el cuerpo del constructor, asigna el título y el año

Algoritmo 1.9 archivo autor.cpp

```
#include <iostream>
#include "autor.h"

Autor::Autor(std::string n, std::string ln, std::string lr) {
    //std::cout << "Construyo un autor con lugar de ";
    //std::cout << "residencia y nacimiento " << std::endl;
    nombre = n;
    lugar_nacimiento = ln;
    lugar_residencia = lr;
}

Autor::Autor(std::string n) {
    //std::cout << "Construyo un autor sin lugar de ";
    //std::cout << "residencia ni nacimiento " << std::endl;
    nombre = n;
    lugar_nacimiento = "";
    lugar_residencia = "";
}

Autor::Autor(Autor& p) {
    //std::cout << "Constructor de copia " << std::endl;
    nombre = p.nombre;
    lugar_nacimiento = p.lugar_nacimiento;
    lugar_residencia = p.lugar_residencia;
}

Autor::~Autor() {
    //std::cout << "Destruyo una persona " << std::endl;
}

void Autor::mostrar() {
    std::cout << "Autor: " << nombre << std::endl;
    std::cout << "Nacio en: " << lugar_nacimiento << std::endl;
    std::cout << "Vive en: " << lugar_residencia << std::endl;
}

void Autor::cambiar_lugar_residencia(std::string lr) {
    lugar_residencia = lr;
}
```

Algoritmo 1.10 archivo novela.h

```
#ifndef Novela_H_INCLUDED
#define Novela_H_INCLUDED
#include "autor.h"

class Novela {
private:
    // Atributos
    std::string titulo;
    int anio;
    Autor autor;

public:
    // Metodos
    Novela(std::string t, int f, Autor a);
    Novela(std::string t, int f);
    ~Novela();

    void mostrar();
    void cambiar_lugar_residencia(std::string lr);
};
#endif // Novela_H_INCLUDED
```

de la novela.

El constructor que no recibe un Autor, como vemos en la línea 10, crea un autor anónimo. El resto de los métodos no presentan nada novedoso.

Ahora veamos en el algoritmo 1.12 un uso de esta clase Novela. En la línea 7 creamos un objeto Novela que se llama *literatura_espaniola* con el título "Lazarillo de Tormes" y año 1554. El autor de esta novela del siglo XVI es desconocido, por lo que no le pasamos ningún autor. En la siguiente línea, creamos un objeto Autor, cuyo nombre es *escritor_terror*, le pasamos el nombre, el lugar de nacimiento y el de residencia, siendo ambos el mismo.

En la línea 9, creamos otro objeto Novela, *leyendo*, con el título, año 1977 y el Autor que acabamos de crear en la línea anterior. Finalmente, mostramos las dos novelas. La salida es la siguiente:

```
El nombre de la novela es: Lazarillo de Tormes
La fecha es: 1554
Autor: Anonimo
Nacio en:
Vive en:
```

Algoritmo 1.11 archivo novela.cpp

```
1  #include <iostream>
2  #include "novela.h"
3
4  Novela::Novela(std::string t, int f, Autor a) : autor(a) {
5      //std::cout << "Construyo un Novela con autor" << std::endl;
6      titulo = t;
7      anio = f;
8  }
9
10 Novela::Novela(std::string t, int f) : autor("Anonimo") {
11     //std::cout << "Construyo un Novela con autor desconocido" << std::endl;
12     titulo = t;
13     anio = f;
14 }
15
16 Novela::~Novela() {
17     //std::cout << "Destruyo un Novela" << std::endl;
18 }
19
20 void Novela::mostrar() {
21     std::cout << "El nombre de la novela es: " << titulo << std::endl;
22     std::cout << "La fecha: " << anio << std::endl;
23     autor.mostrar();
24 }
25
26 void Novela::cambiar_lugar_residencia(std::string lr) {
27     autor.cambiar_lugar_residencia(lr);
28 }
```

Algoritmo 1.12 archivo main.cpp uso Novela

```
1  #include <iostream>
2  #include "novela.h"
3
4  using namespace std;
5
6  int main() {
7      Novela literatura_eapaniola("Lazarillo de Tormes", 1554);
8      Autor escritor_terror("Stephen King", "Maine", "Maine");
9      Novela leyendo("El resplandor", 1977, escritor_terror);
10     literatura_eapaniola.mostrar();
11     leyendo.mostrar();
12     return 0;
13 }
```

```
El nombre de la novela es: El resplandor
La fecha es: 1977
Autor: Stephen King
Nacio en: Maine
Vive en: Maine
```

Tenemos que tener en cuenta que el objeto Autor con los datos de Stephen King está duplicado: uno es el que creamos en el main, el otro, es el que copió al crear la novela *El resplandor*. ¿Qué sucede si creamos otro objeto Novela, con el título Misery? El autor también es Stephen King, por lo que tendríamos una nueva copia con los datos de él. Tengamos en cuenta que Stephen King escribió más de 50 novelas, si creáramos un vector con todas sus novelas, tendríamos más de 50 objetos de la clase Autor con sus datos repetidos.

De lo anterior surgen dos problemas importantes: por un lado, estamos desperdiciando memoria. Debemos tener presente que este ejemplo es una simplificación de la realidad: una clase de tipo Autor tendría muchos atributos, no solo los tres que definimos. Y esta situación sucedería con los demás autores: Vargas Llosa, García Márquez, etc. Por otro lado, se da un inconveniente aún mayor. Supongamos que Stephen King, finalmente, decide irse de Maine porque es un lugar muy peligroso: está lleno de vampiros, brujas, perros rabiosos, fantasmas y gente con poderes mentales destructivos, entre algunas de esas amenazas. Decide mudarse a un lugar tranquilo, como Buenos Aires, para pasar de manera sosegada su senectud. ¿Cómo impacta eso en nuestra aplicación? Deberíamos ubicar

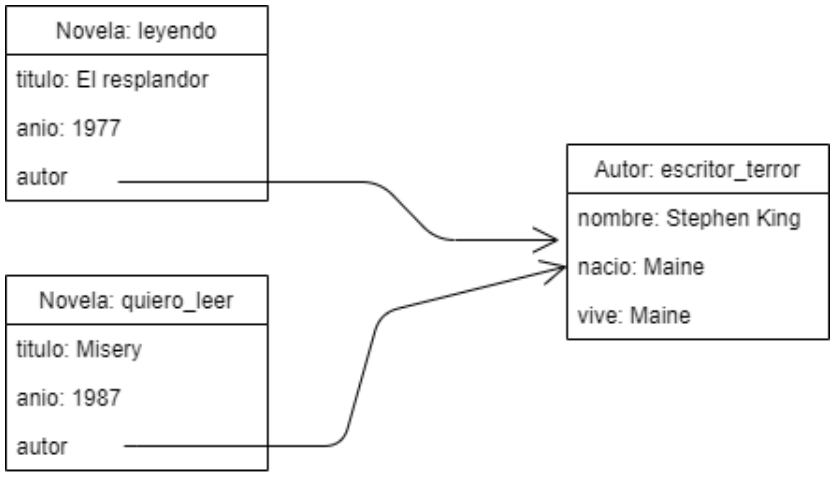


Gráfico 1.17.3: Atributo con referencia a un objeto externo

todos los objetos duplicados de King para cambiar su lugar de residencia, con el riesgo de dejar alguno olvidado en cierto rincón lo que generaría lo que llamamos *inconsistencia*. Tendríamos que King viviría en Buenos Aires para algunos de los objetos Novela y, en Maine, para otros.

La solución a ambos problemas es simple: tengamos un solo objeto con los datos de King, y que cada novela escrita por él, en lugar de tener una copia de sus datos, tenga una referencia o puntero (en este contexto estamos usando la palabra puntero como un sinónimo de referencia) a dicho objeto. De esta forma, en un instante del sistema, un ejemplo de la relación entre los objetos, podría ser como apreciamos en la figura 1.17.3. Hay que tener cuidado que no son diagramas de clases, sino de objetos. Vemos que el atributo autor de dos objetos de la clase Novela hace referencia al mismo escritor.

El código que analizamos anteriormente varía muy poco.

La clase Autor, con sus archivos *autor.h* y *autor.cpp* queda igual. La clase Novela tiene los siguientes cambios:

- En *novela.h* el atributo autor, ahora será un puntero: `Autor* autor;`
- El constructor, también recibe un puntero a Autor
- Además, sacamos el método que cambiaba el lugar de residencia de un Autor, ¿por qué? Autor no es más un objeto que pertenece a Novela, por lo

que no necesitamos indicarle a Novela que cambie alguna característica de un objeto que era suyo. Si necesitamos cambiar algo del Autor lo haremos en la única instancia y quedará actualizado en todos lados.

- En novela.cpp, en el constructor, ya no necesitamos usar inicializadores porque no debemos crear un Autor, solo conectar la referencia con el objeto. De esta forma, el código queda:

```
Novela::Novela(std::string t, int f, Autor* a) {
    titulo = t;
    anio = f;
    autor = a;
}
```

- En el constructor que no tiene Autor, directamente ponemos la referencia en 0: `autor = 0;`
- La decisión anterior fuerza a que cambiemos el método *mostrar*, preguntando si *autor* tiene una referencia válida, lo muestra, de lo contrario, muestra "Anónimo".
- Finalmente, en el *main*, debemos pasar la dirección de *escritor_terror*, con el &:
  `Novela leyendo("El resplandor", 1977, &escritor_terror);`
- Además, creamos una nueva *Novela* de King: *Misery* de 1987.
- También aprovechamos para observar los cambios que se producen en las novelas al modificar un autor que comparten cambiándole el lugar de residencia:
  ```
  escritor_terror.cambiar_lugar_residencia("Buenos Aires");
  leyendo.mostrar();
  ```

Verificamos la salida:

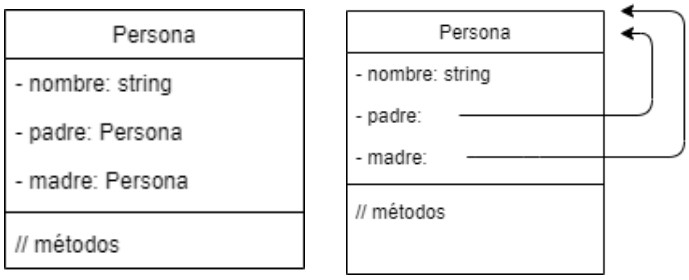
```
---------- Lei en el colegio -------------------
El nombre de la novela es: Lazarillo de Tormes
La fecha es: 1554
Autor: Anonimo
```

```
---------- Estoy leyendo ----------------------
El nombre de la novela es: El resplandor
La fecha es: 1977
Autor: Stephen King
Nacio en: Maine
Vive en: Maine
---------- Pienso leer ------------------------
El nombre de la novela es: Misery
La fecha es: 1987
Autor: Stephen King
Nacio en: Maine
Vive en: Maine
---------- Despues de actualizar datos ---------
El nombre de la novela es: El resplandor
La fecha es: 1977
Autor: Stephen King
Nacio en: Maine
Vive en: Buenos Aires
El nombre de la novela es: Misery
La fecha es: 1987
Autor: Stephen King
Nacio en: Maine
Vive en: Buenos Aires
```

En la salida vemos que solamente al cambiarle los datos (el lugar de residencia) a *escritor_terror*, se actualiza en ambas novelas: *leyendo* y *quiero_leer*.

Si bien este nuevo código representa una mejora sustancial con respecto a nuestra primera aproximación, nos queda observar un detalle más. ¿Qué pasa si el ámbito de nuestro objeto autor termina antes que las novelas? En este ejemplo creamos todos los objetos en el main, por lo tanto, "mueren" todos juntos con el cierre de las llaves. Pero es muy común y recomendable crear los objetos adentro de algún módulo. El problema se da cuando la referencia nos señala un objeto que ya no existe. Para solucionar esto deberíamos crear esos objetos, en este caso los de tipo Autor, de forma dinámica y mantenerlos hasta que dejemos de utilizarlos. La complicación surge porque quizá no los utilicemos de forma directa pero sí a través de otros objetos, como los objetos de la clase *Novela*.

Con esto en mente, los únicos cambios a hacer son los siguientes:



(a) Clase recursiva (b) Clase recursiva con referencias

Gráfico 1.18.1: Clases recursivas

```
   // Creamos un objeto de la clase Autor dinámico y lo manejamos a través
de un puntero
   Autor* escritor_terror = new Autor("Stephen King", "Maine", "Maine");
   // A los objetos Novela no le pasamos una dirección, sino el mismo puntero
a Autor
   Novela leyendo("El resplandor", 1977, escritor_terror);
   Novela quiero_leer("Misery", 1987, escritor_terror);
   // Usamos los objetos que creamos
   // ...
   // Cuando los objetos Novela y Autor ya no los necesitamos, liberamos la
memoria
   delete escritor_terror;
```

1.18. Clases recursivas

Supongamos que queremos implementar una clase Persona, cuyos atributos son: nombre, padre y madre. El atributo nombre será un string pero los atributos padre y madre, también, serán de la clase Persona. El diagrama UML de la figura 1.18.1 representa esta clase. La figura del lado izquierdo muestra de manera explícita sus atributos. La figura del lado derecho lo hace con referencias.

Para implementar esta clase, en un primer intento escribiríamos el siguiente código:

```
class Persona {
private:
    std::string nombre;
```

```
    Persona padre;
    Persona madre;
    ...
};
```

Sin embargo, esto nos daría error dado que la clase *Persona* no está definida para utilizar un objeto de ese tipo, tanto en *padre* como en *madre*. Por ese motivo nos vemos obligados a utilizar referencias (punteros a objetos de tipo *Persona*). Esto sí lo admite.

Por otra parte, definir solo un constructor con tres parámetros (nombre, y punteros a padre y madre) no sería una buena idea, dado que al crear tanto el padre, como la madre, nos pediría, además de sus nombres, los padres de cada uno, es decir, los abuelos del objeto *Persona* que queremos crear. Esto continuaría de manera recursiva sin un punto de parada. Por lo que definimos un nuevo constructor con un solo parámetro: el nombre. En este constructor ponemos en *null* los atributos *padre* y *madre*. En los algoritmos de los recuadros 1.13 y 1.14 están las implementaciones y en el 1.15 hay un ejemplo de su uso, donde tanto *musico* como *esposa* utilizan el constructor de un solo parámetro, y el objeto *hijo*, el de tres.

Algoritmo 1.13 archivo persona.h

```
#include <string>

class Persona {
private:
    // atributos
    std::string nombre;
    Persona* padre;
    Persona* madre;
public:
    // constructor con tres parametros
    Persona(std::string n, Persona* p, Persona* m);
    // constructor con un parametro
    Persona(std::string n);
    // muestra los datos
    void mostrar();
};
```

Algoritmo 1.14 archivo persona.cpp

```
#include "persona.h"
#include <iostream>

using namespace std;

Persona::Persona(std::string n, Persona* p, Persona* m) {
    nombre = n;
    padre = p;
    madre = m;
}

Persona::Persona(std::string n) {
    nombre = n;
    padre = 0;
    madre = 0;
}

void Persona::mostrar() {
    cout << "La persona se llama: " << nombre << endl;
    if (padre)
        cout << "El padre es: " << padre->nombre << endl;
    if (madre)
        cout << "La madre es: " << madre->nombre << endl;
}
```

Algoritmo 1.15 archivo main.cpp - uso clase Persona

```
#include <iostream>
#include "persona.h"

using namespace std;

int main() {
    Persona musico("Johann Sebastian Bach");
    Persona esposa("Ana Magdalena Bach");
    Persona hijo("Johann Christian", &musico, &esposa);
    hijo.mostrar();
    return 0;
}
```

1.19. Resumen del capítulo

- La Programación Orientada a Objetos (POO) facilita la reutilización de código y los cambios en el mismo. Esta última tarea es mal llamada *mantenimiento*, decimos *mal llamada* porque no consiste en mantener el mismo código, sino cambiarlo.

- Algunas características de la POO son:

  - Encapsulamiento: datos y operaciones en un mismo lugar.
  - Ocultamiento de la información: los atributos (datos) deben ser privados.
  - Herencia y polimorfismo: características que veremos en el siguiente capítulo.

- Distinguimos tres etapas:

  - Diseño: donde se produce un TDA, representación abstracta generalmente descripta por diagramas UML.
  - Implementación: se implementa el código del TDA, generando una clase.
  - Uso: se crean objetos de las clases implementadas en la etapa anterior.

- Los atributos deben ser privados. Los métodos, en general, serán públicos, pero pueden existir algunos métodos privados para que no sean utilizados por fuera de la clase. Es una decisión de diseño.

- El puntero this es un puntero que guarda la dirección del propio objeto. Todo objeto tiene un puntero this que se crea de manera automática. Nos sirve para resolver ambigüedades, entre otras cosas.

- Los constructores son métodos especiales que no devuelven nada, ni siquiera *void,* y se llaman en el momento de crear un objeto, ya sea de forma estática como de forma dinámica.

- Por clase, puede haber más de un constructor siempre que difieran en la cantidad o en el tipo de parámetros. Cuando hacemos esto decimos que estamos sobrecargando los constructores. Siempre es recomendable tener un constructor sin parámetros.

- No solo los constructores pueden ser sobrecargados, también cualquier método. También pueden sobrecargarse los operadores.

- Si se maneja memoria dinámica es altamente recomendable escribir un constructor de copia y sobrecargar el operador igual.

- Si se maneja memoria dinámica debería escribirse un destructor que se encargue de liberar la memoria. El destructor no lleva parámetros y es único: no puede sobrecargarse.

- Los atributos estáticos son atributos de la clase y no del objeto. Se utilizan para controlar números de serie correlativos, como números de legajo, etc. Estos atributos deben inicializarse. Los métodos estáticos son necesarios para operar con atributos estáticos.

- Las pre y poscondiciones son comentarios que se colocan al principio de cada método y funcionan como un contrato: las precondiciones son restricciones que debe cumplir el usuario: antes de llamar al método, el estado del objeto debe ser tal o los atributos deben cumplir tal cosa. Las poscondiciones son garantías que da el implementador si se cumplen las precondiciones: luego de ejecutarse el método, el estado del objeto será tal o se devolverá tal cosa.

- Para las clases compuestas debe evaluarse si el objeto contenido es independiente del objeto que lo contiene o no. Si lo fuera, es recomendable utilizar punteros para no duplicar la información y mantenerla de forma consistente.

- Para las clases recursivas (objetos que se incluyen alguno de la propia clase) es necesario usar punteros y prestar atención a los constructores para resolver cualquier recursión infinita.

Cuestionario

a. ¿Qué es un TDA?

b. ¿Qué paradigmas de programación se utilizaron antes de la POO? ¿Cuáles eran sus principales problemas?

c. ¿Qué es una clase?

d. ¿Qué es un objeto?

e. ¿Cuáles son las principales características en la POO?

f. ¿Cómo se debe encarar un problema en la POO?

g. ¿Qué significa public y private?

h. ¿Qué significa static? ¿Desde dónde conviene acceder a un atributo de tipo static? Dar un ejemplo de su uso.

i. ¿Qué es un constructor?

j. ¿Qué es un destructor? ¿Cuándo se debe programar uno?

k. ¿Qué es la sobrecarga de métodos?

l. ¿Qué es el puntero this? ¿Cuándo se debe utilizar?

m. ¿Qué son las Pre y Post condiciones? ¿Para qué sirven?

Ejercicios

a. Diseñar el TDA Fraccion. Una Fraccion tiene que poder inicializarse, simplificarse, sumarse, restarse, multiplicarse y dividirse con otra.

b. Implementar la clase Fraccion diseñada antes. Luego, utilizando esta clase, escribir una calculadora de fracciones.

c. Diseñar e implementar la clase Rectangulo, con atributos base y altura y los métodos para modificar y obtener sus valores, obtener el perímetro y el área.

d. Diseñar e implementar la clase Alimento. Un alimento tiene un nombre y una cantidad de calorías asociada cada 100 gramos.

e. Escribir en una clase Principal que utilice el objeto implementado en el punto anterior.

f. Diseñar e implementar la clase *Partido*. Un *Partido* tiene dos equipos: local y visitante y un resultado (decidir cómo implementar resultado). Un *Equipo* tiene un nombre y un año de fundación. Decidir si cada equipo se guarda adentro del objeto de la clase *Partido* o si solo se tiene una referencia.

g. Generar un vector de objetos de tipo *Partido* y ofrecer un menú para: mostrar los resultados, mostrar el partido con más goles y mostrar el o los equipos con mayor puntaje. El puntaje se calcula por partido: 3 puntos para el ganador, 0 para el perdedor, 1 punto para cada uno en caso de empate.