

Programación Orientada a Objetos

1. Abstracción – Tipo de Dato Abstracto

Seguramente, el lector se encontrará familiarizado con la expresión “tipo de dato”, ya que habrá escrito una serie de programas o aplicaciones en las que ha utilizado diferentes tipos de datos. Por ejemplo, un entero para guardar una edad o una cadena de caracteres para guardar un nombre.

Sin embargo, el término “abstracción”, probablemente, le resulte nuevo. Quizás, esta palabra le represente una imagen de algún cuadro moderno o alguna letra de Spinetta. También, quizás, le recuerde alguna frase peyorativa de su maestra de primaria “estás abstraído”, refiriéndose a que no le estaba prestando atención.

Estas imágenes no tienen que ver con la idea que se le quiere dar al término “abstracción” en informática.

Buscando la definición en el diccionario de la Real Academia Española encontramos:

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 de Borges “Funes el memorioso”, el protagonista (Funes), es una persona que recuerda todo lo que ha vivido con la mayor exactitud. El redactor dice en un párrafo “Sospecho, sin embargo, que no era muy capaz de pensar. Pensar es olvidar diferencias, es generalizar, abstraer.”

Y acá está 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 sólo 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 con dos enfoques distintos:

- 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 contruidos, por lo que se tomarán los recaudos necesarios para que esto no suceda y, por otro lado, no se darán garantías sobre los fallos de la unidad si se detectara 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, sólo 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, ya en el mundo informático. Aunque, se verá en otros cursos, una clase puede ser abstracta en parte o completamente. Si fuera una clase totalmente abstracta diremos que es un TDA.

Un TDA se define indicando las siguientes cosas:

- Nombre (tipo)
- Invariantes
- Operaciones

El nombre nos indica al tipo que nos referimos. Los invariantes nos indican la validez de los elementos que componen el TDA, es decir, qué valores son válidos para conformar un TDA. Las operaciones son las que se necesitan que realice el TDA, se indican con el nombre (o el signo) de la operación, el dominio y el codominio, además, se deben definir las PRE y POST condiciones.

Ejemplo 1:

Nombre TDA: entero

Invariante: $\text{entero} \in \mathbb{Z}$

Operaciones

+	: entero x entero	->	entero
-	: entero x entero	->	entero
*	: entero x entero	->	entero
/	: entero x entero	->	double

Tomemos, por ejemplo, la operación +, significa que, haciendo el producto cartesiano entre dos enteros, nos devuelve otro entero.

En cambio, la operación / nos devuelve un double. Según Meyer, esta operación

se representaría con una flecha tachada, porque no todos los enteros pertenecen al dominio de la operación (o función). Si el segundo parámetro es un cero esta operación no se puede realizar, eso se indica en la PRE condición. La POST condición nos dice qué resultado se obtiene siempre que las PRE condiciones sean cumplidas.

Ejemplo 2:

Nombre TDA: cadena = "c1c2...cN"

Invariante: los ci son caracteres pertenecientes al conjunto Θ .

$$\Theta = \{A..Z\} \cup \{a..z\} \cup \{0..9\} \cup \{ , - , / , (,) , " , ' , ! , = , < , > , ? \}$$

Operaciones

cadena	:		->	cadena
+	:	cadena x cadena	->	cadena
reemplazar	:	cadena x posición x carácter	->	cadena
cadlen	:	cadena	->	entero
valor	:	cadena x posición	->	carácter

Las operaciones las podemos clasificar en tres grupos:

- Constructoras. Es el caso de la primera operación, la cual crea una cadena vacía.
- Modificadoras. Alteran el estado del TDA. Es el caso de las operaciones que están en segundo y tercer lugar. La operación + concatena la cadena a otra, generando una nueva cadena. La operación reemplazar, modifica la cadena, en la posición indicada, colocando el carácter que se le pasa por parámetro.
- Analizadoras. No alteran el estado del TDA, sirven para consultas. La operación cadlen nos indica la longitud de la cadena. La operación valor nos devuelve el carácter que se encuentra en la posición solicitada.

Más adelante se verá que en C++ necesitamos, a menudo, definir operaciones destructoras, son las encargadas, generalmente, de liberar la memoria dinámica requerida. Estas operaciones son modificadoras, ya que destruyen el TDA (cambian su estado).

2. Diseño, implementación y uso de un TDA

En la etapa de diseño no hablamos ni pensamos en su implementación. Ni siquiera se debe 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. Veremos esto con un ejemplo.

DISEÑO

Nombre TDA: Complejo = (real, imaginario)

Invariante: real, imaginario $\in \mathbb{R}$

Operaciones

Complejo	:	$\mathbb{R} \times \mathbb{R}$	->	Complejo
sumar	:	Complejo x Complejo	->	Complejo
restar	:	Complejo x Complejo	->	Complejo
modulo	:	Complejo	->	\mathbb{R}

Para no complicar el TDA se definieron sólo estas cuatro operaciones.

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.

Pre condición: los dos parámetros deben ser números reales.

Post condición: 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.

Pre condición: los dos parámetros deben ser números complejos.

Post condición: devuelve un número complejo. La parte real tendrá el valor de la suma de las partes reales de los dos parámetros, la parte imaginaria el valor de la suma de las partes imaginarias de los dos parámetros.

Operación: restar

Descripción: resta dos números complejos.

Pre condición: los dos parámetros deben ser números complejos.

Post condición: 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 o valor absoluto de un número complejo.

Pre condición: el parámetro debe ser un número complejo.

Post condición: 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.

IMPLEMENTACIÓN 1

Archivo complejo.h

```
#ifndef COMPLEJO_INCLUDED
#define COMPLEJO_INCLUDED

class Complejo
{
private:
    // atributos
    double real;
    double imaginario;

public:
    // constructor con parámetros
    // PRE:      re, im son de tipo double
    // POST:     construye un Complejo, en donde
    //           real = re
    //           imaginario = im
    Complejo (double re, double im);

    // metodo sumar
    // PRE:      c es de tipo Complejo
    // POST:     devuelve un nuevo Complejo, en donde
    //           nuevo.real = real + c.real
    //           nuevo.imaginario = imaginario + c.imaginario
    Complejo sumar(Complejo c);

    // metodo restar
    // PRE:      c es de tipo Complejo
    // POST:     devuelve un nuevo Complejo, en donde
    //           nuevo.real = real - c.real
    //           nuevo.imaginario = imaginario - c.imaginario
    Complejo restar(Complejo c);

    // metodo modulo
    // PRE:
    // POST:     devuelve un double calculado como
    //           raiz(real * real + imaginario * imaginario)
    double modulo();
};

#endif // COMPLEJO_INCLUDED
```

Archivo complejo.cpp

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

// constructor con parametros
Complejo::Complejo(double re, double im)
{
    real      = re;
    imaginario = im;
}

// metodo sumar
Complejo Complejo::sumar(Complejo c)
{
    Complejo aux(0.0, 0.0);
    aux.real      = real + c.real;
    aux.imaginario = imaginario + c.imaginario;
    return aux;
}

// metodo restar
Complejo Complejo::restar(Complejo c)
{
    Complejo aux(0.0, 0.0);
    aux.real      = real - c.real;
    aux.imaginario = imaginario - c.imaginario;
    return aux;
}

// metodo modulo
double Complejo::modulo()
{
    double aux = real*real + imaginario*imaginario;
    return sqrt(aux);
}
```

Analizando la implementación

En primer lugar, por norma, se dividirán en dos archivos distintos lo que es la declaración de la clase, sus atributos y métodos (operaciones) de la definición de las mismas. Si bien todo puede escrito y definido dentro de un mismo archivo es recomendable no hacerlo. También, es recomendable, no definir más de una clase en un mismo archivo. Entonces, en un archivo de extensión .h, que lo llamaremos como el nombre del tipo de dato (pero en minúscula), declaramos la clase, en este caso, la sentencia

class Complejo

junto con las llaves de apertura y cierre (no olvidar el punto y coma luego del cierre de llaves) definen nuestra clase Complejo. En el cuerpo de la clase irán los atributos y los métodos, pero sólo la firma o cabecera.

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 sólo a través de su interfaz (volante, pedales, llave, etc), los atributos se colocan en una sección privada (private). 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 va la cabecera o firma de los métodos, 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, sólo su tipo, se eligió colocarlos para definir las PRE y POST condiciones con los nombres de los parámetros. Por ejemplo, el constructor podría haberse declarado de la siguiente forma:

```
Complejo (double, double);
```

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.

Observando la firma del método sumar, por ejemplo, vemos que sólo tiene un parámetro, cuando necesitamos dos números para poder sumarlos. Sin embargo, el primer parámetro está dado en forma implícita y es el propio objeto.

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

```
#ifndef COMPLEJO_INCLUDED
#define COMPLEJO_INCLUDED
#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 capítulo donde se aborda el tema.

Nota: la parte pública de la clase es lo que se llama la interfaz, porque son los métodos que podrá utilizar el usuario. La firma de los métodos con sus PRE y POST

condiciones debería ser lo único que el usuario debería tener y conocer de la clase.

En el archivo .cpp que, por convención se llama igual que el archivo .h, sólo cambia su extensión, se definen todos los métodos. Se debe notar 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.

En el método sumar, por ejemplo, se crea un número complejo, que es donde se guardará la suma y será el objeto que se devolverá. Se debe observar que, cuando se hace referencia a *real* o *imaginario* a secas, se está refiriendo al primer parámetro que es implícito (o el propio objeto desde donde se llama al método). Para referirnos a los atributos del segundo parámetro (o el único que se pasa entre los paréntesis) debemos indicar el nombre del mismo, en este caso *c*.

Nota: se incluyó el archivo cmath para poder utilizar la función sqrt (raíz cuadrada).

USO DE LA CLASE

Un ejemplo de uso de la clase escrita (es importante pensar que ahora somos usuarios) es el siguiente:

Archivo main.cpp

```
#include <iostream>
#include "complejo"

using namespace std;

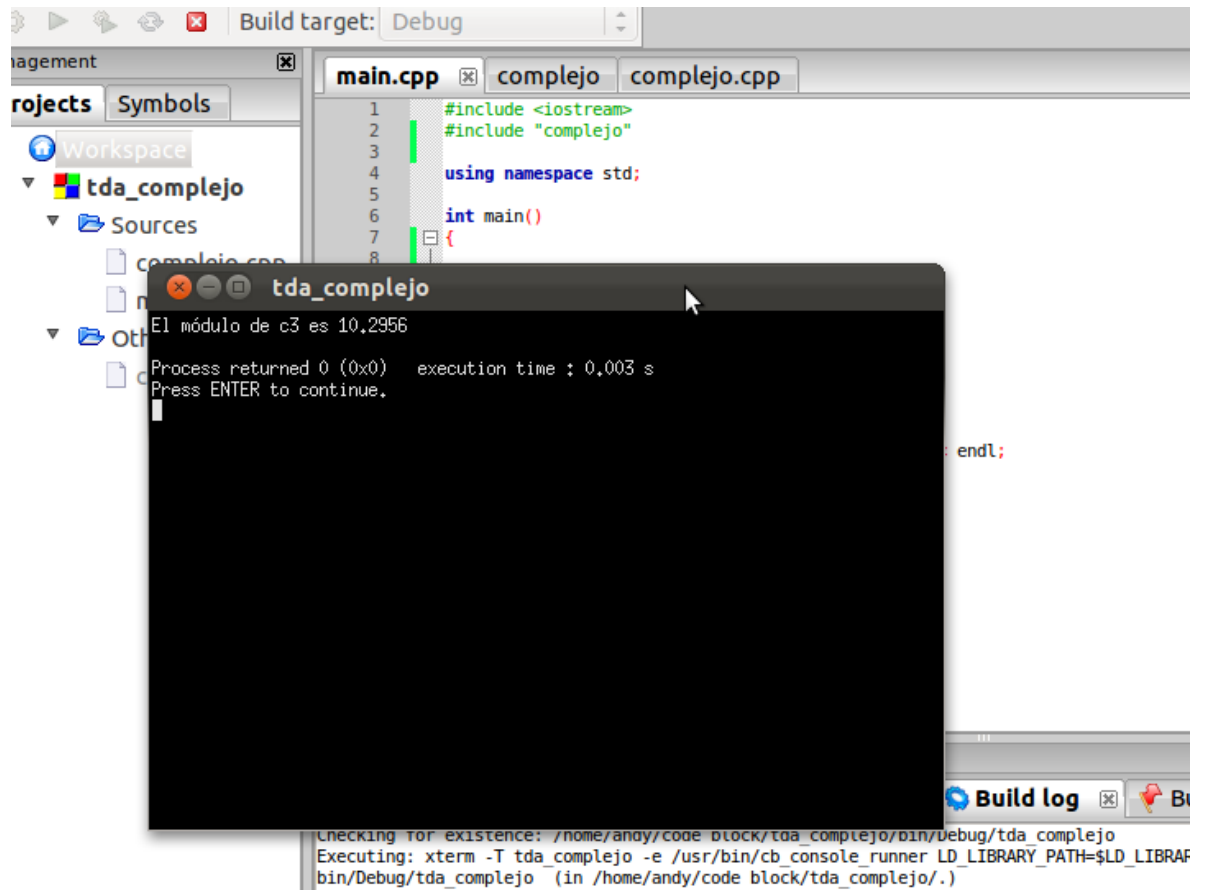
int main()
{
    Complejo c1(5.0, 3.0);
    Complejo c2(4.0, 2.0);
    Complejo c3(0.0, 0.0);

    c3 = c1.sumar(c2);

    cout << "el módulo de c3 es " << c3.modulo() << endl;
    return 0;
}
```

En este ejemplo, creamos tres números complejos, sumamos los dos primeros, guardando el valor en el tercero. Luego se imprime su módulo. Hay que notar cómo se llama al método sumar, desde el primer objeto, siendo el parámetro implícito que se decía.

La salida se puede apreciar en la siguiente figura.



IMPLEMENTACIÓN 2 (con un vector)

Archivo complejo.h

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

```

class Complejo
{
private:
    // atributos
    double z[2];

public:
    // constructor con parámetros
    // PRE:      re, im son de tipo double
    // POST:      construye un Complejo, en donde
    //            real = re
    //            imaginario = im
    Complejo (double re, double im);

    // metodo sumar
    // PRE:      c es de tipo Complejo
    // POST:      devuelve un nuevo Complejo, en donde
    //            nuevo.real = real + c.real
    //            nuevo.imaginario = imaginario + c.imaginario
    Complejo sumar(Complejo c);

    // metodo restar
    // PRE:      c es de tipo Complejo
    // POST:      devuelve un nuevo Complejo, en donde
    //            nuevo.real = real - c.real
    //            nuevo.imaginario = imaginario - c.imaginario
    Complejo restar(Complejo c);

    // metodo modulo
    // PRE:
    // POST:      devuelve un double calculado como
    //            raiz(real * real + imaginario * imaginario)
    double modulo();
};

#endif // COMPLEJO_INCLUDED

```

Archivo complejo.cpp

```

#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;
}

// metodo restar
Complejo Complejo::restar(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;
}

// metodo modulo
double Complejo::modulo()
{
    double aux = z[0] * z[0] + z[1] * z[1];
    return sqrt(aux);
}

```

En esta segunda implementación, en lugar de tener dos atributos: real e imaginario, tenemos un vector de tipo double, este vector lo llamamos z y tiene dos posiciones (la primera para la parte real y la segunda para la parte imaginaria). Lo interesante es notar que, del archivo .h sólo cambia la parte privada y, por supuesto, cambia la definición de los métodos en el archivo .cpp. Sin embargo, la parte pública no se modifica en nada, por lo tanto el usuario podrá seguir ejecutando su aplicación (el archivo main de ejemplo) con esta nueva implementación sin modificar en nada lo escrito.

¿Por qué es esto importante? ¿Qué necesidad puede haber para cambiar una 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 vayan saliendo nuevas versiones de las aplicaciones que utilizamos, con una interfaz más atractiva, amigable, nueva funcionalidad (algunas, no siempre sucede), etc. Sin embargo, las nuevas versiones son capaces de abrir los archivos que hemos creado con versiones anteriores (en general). De lo contrario tendríamos que generar un nuevo archivo por cada nueva versión que se lance. Con las clases y los usuarios de las mismas la idea es similar.

3. Los paradigmas de la programación

Dejaremos de lado, por un momento, los TDA y todo lenguaje de programación, para desarrollar la teoría de la Programación Orientada a Objetos (POO). Pero antes haremos un breve repaso por los paradigmas anteriores.

3.1 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 los “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 complicado. 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.

3.2 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.

La idea de la programación estructurada es resolver los problemas de forma *top-down*, es decir, de arriba hacia abajo, bajo el lema “divide y vencerás”.

El programa principal estará formado por la llamada a las subrutinas (procedimientos o funciones) más importantes, las cuales, a su vez, llamarán a otras subrutinas y, así sucesivamente, hasta que cada subrutina sólo 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 pseudocódigo

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 relleno. Es decir, se van escribiendo las funciones y procedimientos que hagan falta.

3.3 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, inclusive, 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 determinado 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*, *puertas*, 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*.



Imagen 1. Objetos. El mundo está conformado por distintos objetos que interactúan entre sí: semáforos, autos, cámaras, etc.

Es decir, la clase *Celular* será como un molde desde donde nacen los objetos celulares. Entonces, la clase es equivalente al tipo de datos y el objeto a la variable. Por ejemplo:

```
int          x;  
  
(tipo)      (variable)
```

```
Celular      c;  
  
(clase)     (objeto)
```

Por otro lado, tenemos en claro que un objeto *auto* no es lo mismo que un objeto *colectivo* o *ferrocarril*. Pero, todos estos objetos tienen cosas en común. Lo más importante que comparten es que todos sirven como medio de transporte.

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.

Imagen 2. Representación de herencia

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 (notar que en la cajita *Transporte* no se pudo colocar ninguna imagen). Los objetos reales serían los pertenecientes a aquellas clases que hereden de *Transporte*.

Aclaración: 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.

4. Principales conceptos de la POO

Clase

Una clase es un molde de donde se crearán objetos. Es el equivalente al *tipo de dato*. Antes de definir una clase se debe diseñar su TDA, como hemos visto en la sección 2: determinar qué se necesita, qué datos tendrá y qué métodos (funciones) se deberán precisar.

Una clase se puede representar mediante un rectángulo que consiste de tres partes:

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

Imagen 3. Representación de una clase

Por ejemplo, nuestra clase auto se podría modelar con el siguiente gráfico:

Imagen 4. Ejemplo de diseño de una clase

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 unos párrafos más adelante, cuando veamos *encapsulamiento*.

Objeto

Un objeto es una instancia de una clase, el cual tiene valores determinados para sus datos o atributos.

Ejemplo

Imagen 5. Ejemplo de un objeto

Nótese que para determinar un objeto alcanza con su nombre y sus datos. Es decir, no es necesario indicar nada de sus métodos.

Abstracción

Como hemos visto, hay una íntima relación entre una clase y un *Tipo Abstracto de Datos (TDA)*. La abstracción se refiere a conocer algo a través de sus características esenciales y su comportamiento sin saber de su implementación. La abstracción debe estar presente para facilitarnos la vida. La abstracción nos permite utilizar algo, sabiendo cómo se utiliza pero sin preocuparnos cómo está hecho internamente.

Se insiste en los siguientes puntos:

- Conocer el detalle interno de cada artefacto que utilizamos y estar pendientes del mismo nos confundiría y haría nuestra vida muy complicada.
- Por otro lado, nos veríamos tentados a “meter mano” dentro de los artefactos lo cual no sería conveniente. Por algo las garantías se invalidan cuando el usuario intenta arreglar el dispositivo en cuestión por su propia cuenta.

Encapsulamiento

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 y se lo presta.

Éste 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 (interfaz).

Herencia

La herencia la hemos visto con un ejemplo en los medios de transporte. Sin embargo, ahondando un poco más se mostrarán algunos ejemplos mediante diagramas.

Imagen 6. Ejemplo de herencia

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* y *Patente*. Lo mismo sucede con sus métodos.

Polimorfismo

El polimorfismo significa que un objeto puede tener varias (poli) formas (morfo). Es la propiedad que tienen distintos objetos de comportarse de distinta manera ante un mismo mensaje. Si bien no se verá en este curso, se anticipa 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*.

5. ¿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 procedimientos y 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 necesito 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); // Procedimiento que carga el vector  
  
ordenar(vector, n); // Procedimiento que lo ordena  
  
mostrar(vector, n); // Procedimiento que lo muestra
```

Estas funciones tendrían, como parámetros, el vector que deseamos cargar, ordenar y listar; y un valor de n , su tamaño.

Hay que notar que, por ejemplo, el procedimiento *ordenar* no trabajaría de la misma forma en un vector de enteros que de strings, por lo que deberíamos escribir un nuevo procedimiento para las distintas clases de vectores. Éste era uno de los problemas que se mencionaron en la Programación Estructurada: no hay reutilización del código, ya que estos procedimientos serían prácticamente los mismos, sólo que cambiarían sus tipos de datos. Si bien no se verá en este curso, el

polimorfismo nos brinda una solución a este tipo de problemas.

La misma situación, pero encarada en POO, sería de la siguiente manera: 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 método cargar

v.ordenar();        // Se llama al método ordenar

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

Aclaración: si bien el código anterior podría ser de C++ fue pensado como pseudocódigo.

6. Métodos set y get

Unas secciones atrás se recalcó 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 estos dos valores, porque podríamos desear asignar sólo 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, en general, los llamamos *set_nombre_del_atributo*. Por ejemplo *set_real* o *set_imaginario*.

Pero, si quisiéramos consultar esos valores, o imprimirlos, tampoco podríamos acceder a dichos atributos, por lo que deberíamos tener los métodos

para conseguir dichos valores, estos métodos se llaman, por convención, *get_nombre_del_atributo*. Por ejemplo *get_real* o *get_imaginario*.

Nota: podrían llamarse de cualquier manera, se destaca que es sólo por convención.

Obviamente, estos métodos tienen que ser públicos para poder ser accedidos fuera de la clase.

Los métodos *set* colocan el valor del atributo, el cual se deberá pasar por parámetro y no devuelven nada. En cambio los *get* no deben recibir ningún parámetro pero sí deben devolver el valor del atributo.

7. 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* y dejar sus valores por defecto en 0.0. Sin embargo, al crear nuestro constructor con dos parámetros el compilador arrojaría un error si no los indicáramos, teniendo que hacer

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

Sin embargo, podemos sobrecargar el constructor, es decir, escribir otro que no lleve parámetros.

```
// Constructor Sin Parametros - definición
```

```
Complejo::Complejo()  
{  
    real      = 0.0;  
    imaginario = 0.0;  
}
```

Entonces, si el usuario escribiera

```
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.

Por supuesto que es sólo un ejemplo de sobrecarga, a los fines prácticos, para un caso como el expuesto es preferible utilizar 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. Pero, en el caso de no tener parámetros, como en la creación del objeto c1, utiliza los valores por defecto que, en este caso, valen 0.0 para sus dos atributos. En la creación del objeto c3, el valor de 3.5 es tomado para su parte real y, para su parte imaginaria, como falta el valor, toma el valor por defecto que es 0.0.

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 que se desee utilizar el operador + y no un método que se llame sumar, aunque estos números fueran números complejos o de cualquier otro tipo.

Para lograr esto agregamos, a nuestra clase Complejo, en el archivo .h, el siguiente método:

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

Con las mismas PRE y POST condiciones que el método sumar. En nuestro archivo .cpp definimos este método:

```
// operator+  
Complejo Complejo::operator+(Complejo c)  
{  
    Complejo aux(0.0, 0.0);  
    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 en 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.

8. El objeto `this`

this es una referencia (o puntero) a sí mismo que posee todo objeto y se genera de manera automática al crear un objeto.

¿Cuándo es necesario utilizarlo?

Se puede utilizar siempre pero nos veremos obligados a usarlo cuando el compilador no pueda resolver una ambigüedad en los nombres.

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
public Complejo(double real, double imaginario)
{
    this->real      = real;
    this->imaginario = imaginario;
}
```

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

9. Atributos static

Un atributo (o campo) estático es un atributo que no depende de un objeto en particular sino de la clase en sí.

Por ejemplo, se podrán crear varios objetos de una clase *Auto*, en donde, cada uno, tendrá un determinado color. Podría haber varios objetos autos que tuvieran el mismo color. Sin embargo, el dato o atributo *patente* no se podría repetir. Por lo que deseáramos que ese atributo dependiera de otro, como *patente_a_asignar*, en donde, este último atributo no debería pertenecer a un objeto auto en particular, sino a la clase *Auto* directamente para, de ese modo, controlar sus valores desde allí. Por lo tanto, el atributo *patente_a_asignar* debería ser *static*. Cabe resaltar que existe una sola copia por clase de un campo que es *static*.

Imagen 34. Clase y objetos con campos estáticos

En la figura vemos que, cada auto tiene un *color* y una *patente*. Pero deseamos que esa patente conserve 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.

Para que un atributo sea static, sólo 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 decíamos, este 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 a 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. Se mostrará con un ejemplo:

Archivo claseA.h

```
#ifndef CLASEA_INCLUDED
#define CLASEA_INCLUDED

class ClaseA
{
    private:
        // Atributo x (del objeto)
        int x;
        // Atributo y estático (de la clase)
        static int y;

    public:
        // Constructor
        ClaseA(int a);
        // Métodos get (el de y debe ser estático)
        int getx();
        static int gety();
};

#endif // CLASEA_INCLUDED
```

Archivo claseA.cpp

```
#include "claseA"

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

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

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

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



```
        return y;
    }
```

Archivo main.cpp

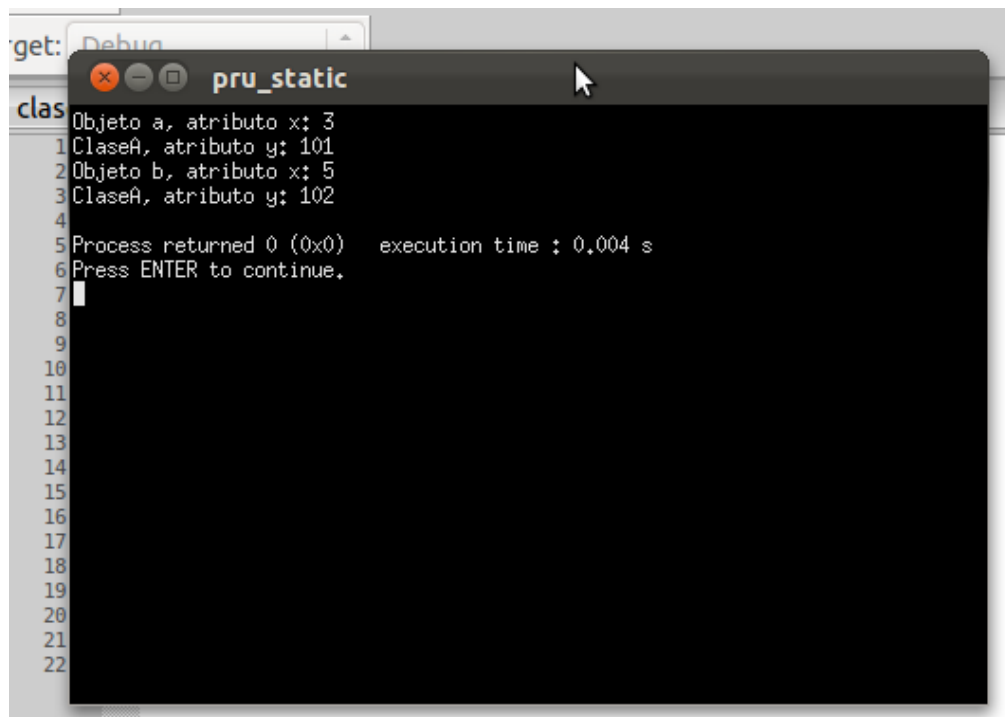
```
#include <iostream>
#include "claseA"

using namespace std;

int main()
{
    ClaseA a(3);
    cout << "Objeto a, atributo x: " << a.getx() << endl;
    // Se llama al método gety desde el objeto
    cout << "ClaseA, atributo y: " << a.gety() << endl;

    ClaseA b(5);
    cout << "Objeto b, atributo x: " << b.getx() << endl;
    // Se llama al método gety desde la clase
    cout << "ClaseA, atributo y: " << ClaseA::gety() << endl;

    return 0;
}
```



```
get: Debug
clas
Objeto a, atributo x: 3
1 ClaseA, atributo y: 101
2 Objeto b, atributo x: 5
3 ClaseA, atributo y: 102
4
5 Process returned 0 (0x0)   execution time : 0.004 s
6 Press ENTER to continue.
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

Analizando el código anterior, se define una clase con nombre ClaseA, en un

archivo .h, como siempre. Esta clase tiene un atributo *x*, que pertenecerá a cada objeto en particular. Pero, además, tendrá un atributo *y* que pertenecerá a la clase, porque es estático.

Supongamos que deseamos que los objetos se vayan creando con un número de serie que empieza en 101, etc. Debemos inicializar el atributo *y*, como se hace en el archivo `claseA.cpp`, se debe notar que esta inicialización está fuera de cualquier método.

En el constructor nos encargamos de incrementar en uno esta variable. Es decir, cada vez que se cree un objeto tipo `ClaseA` se incrementará este contador.

En el archivo `main`, creamos dos objetos de este tipo, 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 objeto, como se hace en el primer caso, como desde su clase, como en el segundo. Se prefiere utilizarlo en esta última forma, ya que, en el primer caso, da la sensación que el atributo es un valor perteneciente al objeto en sí y no a su clase.

10.Pre y Post condiciones

Es muy sano estipular, al principio de cada método, las Pre y Post Condiciones. ¿Qué son? ¿Por qué son necesarias?

Las Pre y Post condiciones son comentarios. En una Pre Condición decimos en qué estado debe estar el objeto para llamar a determinado método y cuáles son los valores válidos de sus parámetros. En la Post Condición 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 dos razones:

- 1) 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.
- 2) 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 pre condiciones, garantizo el resultado”, por lo que son un buen elemento

para determinar responsabilidades. ¿Por qué falló el sistema? ¿Se cumplió la pre condición? Si la respuesta es **sí**, el culpable es el implementador. En cambio, si la respuesta es **no**, el usuario de la clase.

11. Destructores

Los destructores se deben programar para liberar recursos, como cerrar archivos, liberar memoria dinámica utilizada, etc. De lo contrario, no es necesario definirlos, ya que el lenguaje provee destructores de oficio. En el ejemplo de la clase Complejo, que estuvimos viendo, no era necesario liberar ningún recurso, por este motivo no se programó ningún destructor.

Los destructores, al igual que los constructores, deben tener el mismo nombre que la clase, pero con un signo ~ adelante del nombre. Además, no pueden llevar parámetros ni pueden ser sobrecargados (una sobrecarga no sería permitida por el lenguaje ya que se estaría repitiendo la firma del método, aparte de que no tendría ningún sentido).

Es importante recalcar que los destructores no deben ser llamados en forma explícita, se llaman automáticamente cuando se termina el ámbito en donde el objeto fue definido o, cuando se ejecuta la instrucción delete, en el caso de haber sido creado el objeto con el operador new, utilizando memoria dinámica.

Ejemplo:

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

void funcion()
{
    Clase o1;                // Se crea un objeto de tipo Clase
    Clase *ptr = new Clase(); // Se crea un objeto dinámico tipo Clase
}
```

```

...
delete ptr; // Se debe liberar 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 tamano; // longitud del vector
        int *datos; // datos que habrá en el vector

    public:
        // constructores
        Vector(int tam);

        // otros métodos
        ...

        // destructor
        ~Vector();
};

// Definición de los métodos
// Constructor
Vector::Vector(int tam)
{
    tamano = tam;
    // Se solicita memoria dinámica para guardar los datos
    datos = new int[tamano];
}

Vector::~~Vector()
{
    // Se libera la memoria solicitada
    delete []datos;
}

```

12. Constructor de copia

¿Qué es el constructor de copia y por qué se necesita?

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:

```
void funcion(Clase o)
{
    // sentencias de la función
    ...
}
```

```
Clase o1;
funcion(o1);
Clase o2 = o1;
```

En la penúltima línea del ejemplo anterior, se llama a la función definida más arriba, a la que se le pasa un objeto por parámetro. Dentro de la función, se debe trabajar con este objeto pero, este objeto está pasado por valor, es decir que se crea, internamente, una copia del mismo, llamando al constructor de copia.

En la última línea, 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?

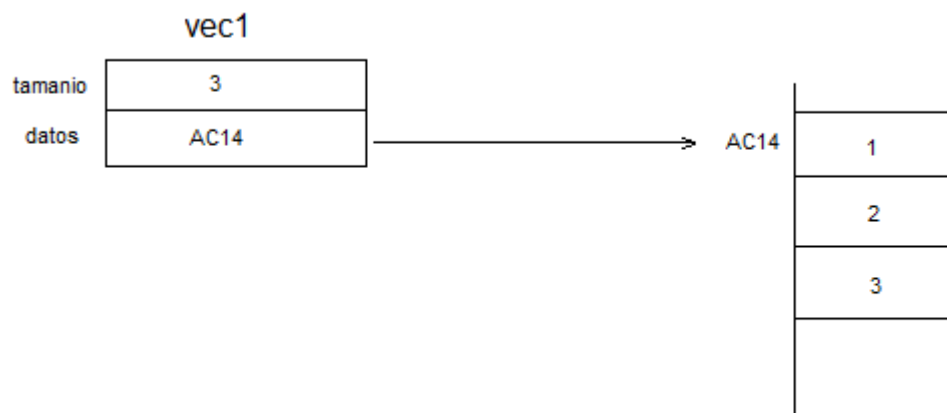
El problema surge cuando se utiliza memoria dinámica. Volvamos al ejemplo de la clase *Vector*. Esta clase *Vector* tiene dos atributos: un entero que indica su tamaño y un puntero, a una zona de memoria donde estarán los datos en sí.

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 un parámetro 3.

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

Luego, con otros métodos 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 *tamano* quedará con el valor 3 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. Lo único que tenemos en el objeto *vec1* que se relaciona con estos datos es la dirección de comienzo.



Ahora bien, cuando otro objeto, supongamos *vec2* copie de *vec1*, copiará todos los valores de sus 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.

Conclusión, si ejecutamos la siguiente sentencia:

```
Vector vec2 = vec1;
```

Nuestro nuevo objeto, *vec2*, tendrá el valor de 3 en *tamano*, lo cual es correcto y, en *datos*, tendrá AC14, es decir que 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 se 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.

Conclusión, en los casos en que manejemos atributos que utilicen memoria dinámica, es necesario que programemos un constructor de copia.

El primer parámetro del constructor de copia (y, en general, el único) debe ser una referencia a un objeto de la misma clase que estamos programando, ya 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.

Ejemplo:

```
Vector::Vector(const Vector &vec)
{
    // los tamaños deben ser iguales
    tamaño = vec.tamaño;
    // Se solicita nueva memoria
    datos = new int[tamaño];
    // Se copian los valores a una nueva porción de memoria
    for (int i = 0; i < tamaño; i++)
        datos[i] = vec.datos[i];
}
```

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

13. Sobrecarga del operador =

Los problemas indicados con el constructor de copia, se repiten cuando se utiliza el

operador =.

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, se debería, también, redefinir al operador = para poder utilizarlo sin que se produzcan efectos no deseados.

Ejercitación

1. ¿Qué es un TDA?
2. Diseñar el TDA Fraccion. Una Fraccion tiene que poder inicializarse, simplificarse, sumarse, restarse, multiplicarse y dividirse con otra.
3. ¿Qué paradigmas de programación se utilizaron antes de la POO?
4. ¿Cuáles eran sus principales problemas?
5. ¿Qué es una clase?
6. ¿Qué es un objeto?
7. ¿Cuáles son las principales características en la POO?
8. ¿Cómo se debe encarar un problema en la POO?
9. ¿Qué significa public y private?

10. ¿Qué significa static? ¿Desde dónde conviene acceder a un atributo de tipo static? Dar un ejemplo de su uso.
11. ¿Qué es un constructor?
12. ¿Qué es un destructor? ¿Cuándo se debe programar uno?
13. ¿Qué es la sobrecarga de métodos?
14. ¿Qué es el objeto this?
15. ¿Qué son las Pre y Post condiciones? ¿Para qué sirven?
16. Escribir la clase *Rectangulo*, con atributos *base* y *altura* y los métodos para modificar y obtener sus valores, obtener el perímetro y el área.
17. Escribir la clase *Alimento*. Un *alimento* tiene un nombre y una cantidad de calorías asociada cada 100 gramos.
18. Escribir en una clase *Principal* la utilización de los objetos creados en los puntos anteriores.
19. Escribir la clase *Fraccion* diseñada antes. Luego, utilizando esta clase, escribir una calculadora de fracciones.