TDA

- Materia Algoritmos y Estructuras de Datos
- Cátedra Schmidt
- Implementación con TDA

Historia

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.

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 y 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.

Programación con TDA

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 con TDA.

Este paradigma intenta modelar el mundo real, en el cual nos encontramos con toda clase de TDAs que se comunican entre sí y reaccionan ante determinados estímulos o mensajes.

Los TDAs están conformados por propiedades o atributos, inclusive, estos atributos pueden ser, a su vez, otros TDAs. 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.

El primer paso hacia la programación de TDAs es la abstracción.

Tipo de Dato Abstracto

Definición: Un Tipo de Dato Abstracto (TDA) es un modelo que define **valores** y las **operaciones** que se pueden realizan sobre ellos. Y se denomina **abstracto** ya que la intención es que quien lo utiliza, no necesita conocer los detalles de la **representación interna** o bien el **cómo** están implementadas las operaciones.

En general, con la mayoría de los dispositivos que usamos sucede que los utilizamos sin saber qué mecanismos internos se están accionando en cada instante ni que partes se ponen en actividad en cada operación.

Por ejemplo, sabemos cual es efecto del botón cambiar de canal del control remoto del televisor, peor no conocemos los mecanismos internos del aparato.

A los efectos para los cuales lo queremos (pasar de un canal a otro), sólo interesa qué hace, no cómo lo hace.

Sabemos cual es el efecto que se produce ante determinado estímulo (o cual es la respuesta o comportamiento ante un mensaje) pero no sabemos cómo está implementado el aparato.

Conocemos lo que está accesible, y no pensamos en cada detalle de lo que está sucede “por debajo” de la interfaz (interfaz es el sistema de comunicación con el usuario).

Nosotros desarrollaremos tipos de datos abstractos (TDA) partiendo de esas consideraciones.

Definición según diccionario

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.

Entonces **¿qué es un tipo de dato abstracto?**

La idea general que se concentra sobre las cualidades o aspectos esenciales de algún objeto del mundo real e ignora las propiedades accidentales. Es un mecanismo de descripción de alto nivel que, que modela un concepto y luego se implementa con una clase.
La abstracción es la capacidad de encapsular y aislar la información del diseño, de la implementación y de la ejecución.

Un TDA se define indicando las siguientes cosas:

- Nombre (tipo de dato)
- Invariantes
- Operaciones y axiomas

Nombre: El nombre nos indica al tipo que nos referimos, es el nombre del tipo de dato y debe ser unico en nuestro espacio de nombre.

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, los parámetros y el retorno, además, se deben definir las PRE y POST condiciones, que son los axiomas.

Ejemplo 1

- Nombre: Entero
- Invariante: Entero ϵ Z
- Operaciones
  - $+$: entero x entero $\rightarrow$ entero
  - - : entero x entero $\rightarrow$ entero
  - * : entero x entero $\rightarrow$ entero
  - / : entero x entero $\rightarrow$ double

Descriptivo

Los objetos de tipo Entero pertenecen a Z

Ejemplo 2

- Nombre: Cadena.

Descriptivo

- Invariante: *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 : → cadena
  - + : cadena x cadena → cadena
  - reemplazar : cadena x posición x carácter → cadena
  - cadlen : cadena → entero
  - valor : cadena x posición → carácter

Indicamos qué condición deben verificar los objetos de tipo Cadena

Programación a alto nivel

1. Resolución del problema
2. Diseño
3. Abstracción de la realidad.
4. Encapsulamiento
5. Convención de nomenclatura de la cátedra
6. Implementación en C++
7. Conjunto completo de operaciones
8. Pre y post condición. Comentarios.
9. Objetos estáticos vs objetos dinámicos
10. Robustez
11. Desarrollo modularizado
12. Testeo. Main de prueba independiente
13. Desarrollo mantenible, extensible y optimizado
14. Campos estáticos
15. Sobrecarga de operadores
16. Sobrecarga de métodos
17. Comparativa de solución de TDA y su correspondiente en C.
18. Ejercicio de Parcial

Resolución del problema

La resolución de problemas es una habilidad crucial en el desarrollo de software, ya que los desarrolladores se enfrentan a desafíos técnicos y lógicos constantemente durante el proceso de construcción de software. La capacidad para resolver estos problemas de manera eficiente y efectiva es fundamental para el éxito en este campo. El objetivo es presentar una solución completa, siguiendo una metodología de manera ordeanda.

Diseño

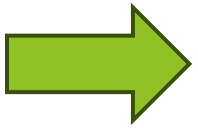
Otra forma habitual de diseñar un TDA es utilizando una simplificación de la notación UML (Lenguaje de Modelado Unificado). En UML Una clase se representa mediante un rectángulo que consiste de tres partes:

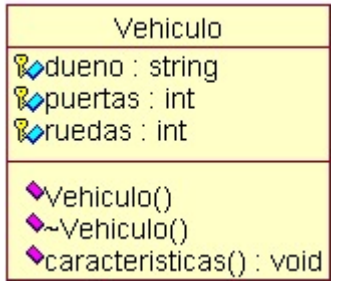
- El nombre de la clase
- Sus datos (atributos)
- Sus métodos (funciones u operaciones)

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

Abstracción

Separar, por medio de una operación intelectual, las cualidades de un objeto para considerarlas para resolver el problema aisladamente.





Encapsulamiento

Con la finalidad de proteger al TDA 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 practica de TDA: colocar los datos de manera inaccesibles desde el exterior y poder trabajar con ellos mediante métodos estipulados a tal fin (interfaz).

Métodos get y set

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 setNombreDelAtributo. Por ejemplo setReal o setImaginario.

```
11 class Complejo {
12 private:
13     double real;
14     double imaginario;
15
```

```
20     double getImaginario();
21     double getReal();
22
23     void setImaginario(double imaginario);
24     void setReal(double real);
25 };
```

Convención de nomenclatura de la cátedra

- Clases: Los nombres de clases siempre deben comenzar con la primera letra en mayúscula en cada palabra, deben ser simples y descriptivos. Se concatenan todas las palabras. Ejemplo: Coche, Vehiculo, CentralTelefonica.
- Métodos: Deben comenzar con letra minúscula, y si está compuesta por 2 o más palabras, la primera letra de la segunda palabra debe comenzar con mayúscula. De preferencia que sean verbos. Ejemplo: arrancarCoche(), sumar().
- Objetos: las objetos siguen la misma convención que los métodos. Por Ejemplo: alumno, padronElectoral.
- Constantes: Las variables constantes o finales, las cuales no cambian su valor durante todo el programa se deben escribir en mayúsculas, concatenadas por "_". Ejemplo: ANCHO, VACIO, COLOR_BASE.
- Comentar el código. Todos los tipos, métodos y funciones deberían tener sus comentarios en el .h que los declara.

Aplicación en C++

- El lenguaje utilizado es C++, esto quiere decir que se debe utilizar siempre C++ y no C, por lo tanto una forma de darse cuenta de esto es no incluir nada que tenga .h, por ejemplo #include <iostream> .
- No usar sentencias 'using namespace' en los .h, solo en los .cpp. Por ejemplo, para referenciar el tipo string en el .h se pone std::string.
- No usar 'and' y 'or', utilizar los operadores '&&' y '||' respectivamente.
- Compilar en forma ANSI. Debe estar desarrollado en linux con eclipse y g++. Utilizamos el estándar C++98.
- Chequear memoria antes de entregar. No tener accesos fuera de rango ni memoria colgada.
- Si el trabajo práctico requiere archivos para procesar, entregar los archivos de prueba en la entrega del TP. Utilizar siempre rutas relativas y no absolutas.
- No inicializar valores dentro del struct o .h.
- Utilizar siempre {} en las estructuras de control

Implementación en C++

```
 2⊕ * Complejo.h
 7
 8 #ifndef COMPLEJO H
 9 #define COMPLEJO H
10
11⊖class Complejo {
12 private:
13     double real;
14     double imaginario;
15
16 public:
17     Complejo();
18     virtual ~Complejo();
19
20     double getImaginario();
21     double getReal();
22
23     void setImaginario(double imaginario);
24     void setReal(double real);
25
26 };
27
28 #endif /* COMPLEJO H  */
```

```
 9
10⊖Complejo::Complejo() {
11     this->imaginario = 0;
12     this->real = 0;
13 }
14
15 Complejo::~Complejo() {}
16
17
18⊖double Complejo::getImaginario(){
19     return imaginario;
20 }
21
22⊖void Complejo::setImaginario(double imaginario) {
23     this->imaginario = imaginario;
24 }
25
26⊖double Complejo::getReal() {
27     return real;
28 }
29
30⊖void Complejo::setReal(double real) {
31     this->real = real;
32 }
```

Constructores

Un constructor es un metodo cuya misión es inicializar un objeto de una clase. En el constructor se asignan los valores iniciales del nuevo objeto.

Posee el mismo nombre de la clase a la cual pertenece y no puede devolver ningún valor (ni siquiera se puede especificar la palabra reservada void). Por ejemplo.

```
9
10 Complejo::Complejo() {
11     this->imaginario = 0;
12     this->real = 0;
13 }
14
```

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.

Puntero this

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

¿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.

```
10 Complejo::Complejo() {
11     this->imaginario = 0;
12     this->real = 0;
13 }
```

```
10 Complejo::Complejo() {
11     imaginario = 0;
12     real = 0;
13 }
```

```
10 Complejo::Complejo(double imaginario) {
11     this->imaginario = 0;
12     real = 0;
13 }
14
```

Conjunto completo de operaciones

Debe tener un conjunto completo de operaciones y sus axiomas asociados. Ya que no hay posibilidad de operar con el por fuera de los métodos, gracias al encapsulamiento. Aquí hay algunas razones por las cuales esto es importante:

Consistencia y coherencia: Definir un conjunto completo de operaciones y axiomas ayuda a garantizar que el TDA sea coherente en su funcionamiento y consistente en su comportamiento. Esto hace que sea más fácil de entender y utilizar correctamente.

Claridad y comprensión: Al especificar todas las operaciones disponibles y sus propiedades asociadas, se proporciona una guía clara sobre cómo interactuar con el TDA. Esto mejora la comprensión de los usuarios sobre cómo utilizar el TDA de manera efectiva y cómo se comportará en diferentes situaciones.

Previsibilidad y fiabilidad: Al tener axiomas bien definidos para cada operación, se establece un conjunto de reglas que rigen el comportamiento del TDA. Esto hace que el comportamiento del TDA sea predecible y confiable, lo que facilita el diseño y la depuración del código que lo utiliza.

Facilita la verificación y la validación: Tener un conjunto completo de operaciones y axiomas facilita la verificación y la validación del TDA. Los desarrolladores pueden realizar pruebas exhaustivas para garantizar que el TDA cumpla con sus especificaciones y que funcione correctamente en una variedad de situaciones.

Facilita la interoperabilidad: Al definir claramente las operaciones y sus propiedades, se facilita la interoperabilidad entre diferentes implementaciones del mismo TDA. Esto permite a los desarrolladores intercambiar fácilmente implementaciones alternativas sin preocuparse por la compatibilidad o el comportamiento inesperado.

En resumen, especificar un conjunto completo de operaciones y axiomas es esencial para garantizar la coherencia, la fiabilidad y la comprensión del TDA, lo que a su vez facilita su uso, desarrollo y mantenimiento.

Métodos

- Constructores
- Destructor
- Gets y sets
- Métodos
- Sobrecarga de métodos

Pre y post condición. Comentarios.

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.

Objetos estáticos vs objetos dinámicos

En C++, los objetos estáticos y dinámicos se refieren a cómo se asigna y se maneja la memoria para los objetos durante la ejecución del programa.

Los objetos estáticos:
1) se crean en tiempo de compilación y se almacenan en una región de memoria llamada Stack.
2) Su ciclo de vida es el mismo que el del programa y se asignan memoria en el momento en que se declara la variable.
3) Tienen un alcance dentro de la función, clase o archivo donde se declaran.
4) Los objetos estáticos se inicializan solo una vez, antes de que comience la ejecución del programa.

Los objetos dinámicos:
1) Se crean en tiempo de ejecución utilizando operadores de asignación de memoria, como new en C++.
2) Se almacenan en una región de memoria llamada "heap".
3) El ciclo de vida de un objeto dinámico no está limitado por el alcance de la función o el bloque donde se crea; sigue existiendo hasta que se elimina explícitamente.
4) Los objetos dinámicos deben eliminarse manualmente utilizando el operador delete cuando ya no sean necesarios para evitar fugas de memoria.

En resumen, los objetos estáticos se asignan y se liberan automáticamente en tiempo de compilación y su ciclo de vida está vinculado al del programa, mientras que los objetos dinámicos se crean y se liberan explícitamente durante la ejecución del programa y tienen un ciclo de vida más flexible.

cpp Copy code

```cpp
#include <iostream>
using namespace std;

class Ejemplo {
public:
    static int contador;
    Ejemplo() { contador++; }
    ~Ejemplo() { contador--; }
};

int Ejemplo::contador = 0;

int main() {
    Ejemplo objeto1;
    Ejemplo objeto2;
    cout << "Contador de objetos: " << Ejemplo::contador << endl;
    return 0;
}
```

cpp Copy code

```cpp
#include <iostream>
using namespace std;

class Ejemplo {
public:
    int dato;
    Ejemplo(int d) : dato(d) {}
    ~Ejemplo() { cout << "Destructor llamado" << endl; }
};

int main() {
    Ejemplo* objeto1 = new Ejemplo(10);
    cout << "Dato: " << objeto1->dato << endl;
    delete objeto1;
    return 0;
}
```

```cpp
//Objeto estatico
Complejo complejoEstatico(2, 3);

//Objeto dinamico
Complejo * complejoDinamico = new Complejo(2, 3);
delete complejoDinamico;
```

Robustez

La robustez en el contexto del desarrollo de software se refiere a la capacidad de un sistema para manejar situaciones inesperadas o incorrectas de manera adecuada y predecible. Un software robusto es aquel que puede resistir y recuperarse de errores, fallos o entradas no válidas sin que esto cause un colapso total del sistema.

La robustez se logra mediante la implementación de mecanismos que permiten detectar y manejar excepciones, errores o condiciones inesperadas de manera elegante y segura. Esto puede incluir la validación de entradas, el manejo de errores, la implementación de controles de flujo adecuados y la redundancia de datos, entre otras técnicas. Por mas que la pre condición aclare un estado como valido, es correcto verificarlo para hacer robusto el software.

Algunos aspectos importantes de la robustez en el desarrollo de software incluyen:

Validación de entradas o parámetros: Verificar que las entradas proporcionadas por el usuario o por otros sistemas sean válidas y estén dentro de los límites esperados antes de procesarlas.

Manejo de errores: Implementar mecanismos para detectar, registrar y manejar errores de manera adecuada, proporcionando mensajes de error claros y útiles para guiar al usuario sobre cómo resolver el problema.

Tolerancia a fallos: Diseñar el software de manera que pueda seguir funcionando de manera aceptable incluso cuando ocurran fallos o condiciones inesperadas, evitando que un error aislado provoque una falla catastrófica en todo el sistema.

Recuperación elegante: Implementar estrategias de recuperación que permitan al sistema volver a un estado seguro y funcional después de un fallo, minimizando el impacto en la experiencia del usuario y garantizando la integridad de los datos.

Pruebas exhaustivas: Realizar pruebas exhaustivas para identificar y corregir posibles puntos débiles en el sistema, garantizando que el software sea robusto en una amplia variedad de situaciones y condiciones.

En resumen, la robustez es una cualidad deseada en el desarrollo de software que garantiza que el sistema pueda manejar errores y situaciones inesperadas de manera adecuada, manteniendo su funcionamiento y proporcionando una experiencia de usuario satisfactoria incluso en circunstancias adversas.

Testeo

Para terminar de cerrar la etapa de desarrollo de un TDA hay que hacer una TDA de testeo o un main de prueba.

Desarrollo mantenible, extensible y optimizado

La modularización es un principio de diseño de software que implica dividir un sistema en módulos o componentes más pequeños y manejables. Estos módulos son unidades independientes que realizan funciones específicas y están diseñados para ser intercambiables y reutilizables. Algunas características importantes de la modularización incluyen:

División en TDAs: La modularización implica dividir un sistema en TDAs más pequeños y cohesivos. Cada TDA tiene una responsabilidad única y realiza una función específica dentro del sistema.

Interfaz bien definida: Cada módulo define una interfaz clara y bien definida que especifica cómo interactuar con él. Esta interfaz proporciona una capa de abstracción que separa la implementación interna del módulo de su uso externo, permitiendo que los TDAs se comuniquen de manera efectiva sin exponer sus detalles internos.

Reutilización: Los TDAs modularizados son unidades independientes que pueden ser reutilizadas en diferentes partes del sistema o en otros proyectos. Esto promueve la reutilización del código, reduce la duplicación y facilita la creación de sistemas más flexibles y mantenibles.

Acoplamiento bajo y cohesión alta: La modularización busca minimizar el acoplamiento entre los módulos, lo que significa que los TDAs deben ser independientes entre sí y tener pocas dependencias externas. Al mismo tiempo, se busca maximizar la cohesión dentro de cada TDA, asegurando que las partes relacionadas estén agrupadas de manera lógica y que cada módulo tenga una responsabilidad única y bien definida.

Facilidad de mantenimiento: Al dividir un sistema en TDAs más pequeños y cohesivos, la modularización facilita el mantenimiento del código. Los cambios o actualizaciones pueden realizarse de forma más rápida y segura, ya que afectan solo a los módulos específicos relacionados con la funcionalidad en cuestión.

Campos estáticos

Un atributo o método estático es uno que no depende de un objeto en particular sino de la clase en sí.

Por ejemplo, se podrán crear varios objetos de una clase Complejo, en donde, cada uno, tendrá un determinado numero. Sin embargo, la suma de dos complejos cualesquiera puede ser que dependa de la clase. Por lo tanto, el método sumar debería ser static.

Cabe resaltar que existe una sola copia por clase de un campo que es static.

```
 9 #define COMPLEJO H
10
11 class Complejo {
12 public:
13     static int id;
14     static Complejo * sumar1(Complejo * c1);
15
16 private:
17     double parteReal;
18     double parteImaginaria;
19     char * letra;
20
21 public:
22
```

```
34     //Correcto
35     Complejo::sumar1(&complejo3);
36     Complejo::id = 1;
37
38     //Funciona
39     complejo.sumar1(&complejo3);
40
```

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);
// 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.

Sobrecarga de metodos

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(); o Complejo c(double, double);

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:

Complejo::Complejo(double re = 0.0, double im = 0.0)

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.

Comparativa de solución de TDA y su correspondiente en C.

```c
 * ComplejoEnC.h
 * ComplejoEnC.h

#ifndef COMPLEJOENC H
#define COMPLEJOENC H

typedef struct {
    double real;
    double imaginaria;
} ComplejoEnC;

ComplejoEnC construirComplejoEnC();
ComplejoEnC sumar(ComplejoEnC complejo1, ComplejoEnC complejo2);
void setParteReal(ComplejoEnC &complejo, double real);

#endif /* COMPLEJOENC H  */


#endif /* COMPLEJOENC H  */
```

```c
#ifndef COMPLEJO H
#define COMPLEJO H

class Complejo {
public:
    static int id;
    static Complejo * sumar1(Complejo * c1);

private:
    double parteReal;
    double parteImaginaria;
    char * letra;

public:

    /**
     * Pre: -
     * Post: construye el complejo y deja el complejo en el valor (0,0)
     */
    Complejo();

    Complejo(Complejo &complejo);
```

Ejercicio de Parcial

2) Diseñar la especificación e implementar los TDAs modelen una **Cadena**, cumpliendo las siguientes características:

Una **Cadena** está compuesta por una secuencia de **Eslabones**. Toda **Cadena** tiene al menos un **Eslabón**. Cada **Eslabón** posee largo y ancho. Todos los eslabones de la **Cadena** deben tener el mismo ancho, pero el largo puede diferir.
Se debe poder agregar o retirar **Eslabones** de la **Cadena**.
Se requiere conocer la longitud total de la **Cadena**.

Fin