Unidad I: Tipo de Dato Abstracto (TDA)

Introducción

Suponga que debe resolver el problema de **verificar si dado 2 números complejos su suma y multiplicación son iguales**.

Para resolver dicho problema se debe recordar primero qué es un número complejo (tomado de Wikipedia):

> Describe la suma de un **número real** y un **número imaginario** (que es un múltiplo real de la unidad imaginaria, que se indica con la letra i). Los números complejos se utilizan en todos los campos de las matemáticas, en muchos de la física (y notoriamente en la mecánica cuántica) y en ingeniería, especialmente en la electrónica y las telecomunicaciones, por su utilidad para representar las ondas electromagnéticas y la corriente eléctrica.
>
> Definiremos cada complejo z como un par ordenado de números reales (a, b) ó (Re(z), Im(z)), en el que se definen las siguientes operaciones:
>
> * Suma
>
> (a, b) + (c, d) = (a+c) +\; (b+d)i
>
> * Multiplicación
>
> (a, b) * (c, d) = (ac - bd) + (ad + cb)i
>
> * Igualdad
>
> (a, b) = (c, d) ←→ a = c y b = d

Claramente un número complejo requiere una estructura de datos simple, como un registro o estructura, para manejar el par ordenados de número reales como su parte real e imaginaria. Para resolver el problema, luego es necesario calcular la suma y la resta de los dos números y luego ver si estos resultados son el mismo número complejo.

Una posible solución en C++ es el siguiente:

```
#include <iostream>
using namespace std;
struct Complejo
 {
   double PReal;
```

```
   double PImaginaria;
 };

int main()
{
    Complejo A,B,C,D;
      cout<<"**Introduzca el primer numero COMPLEJO"<<endl<<"Parte real:"<<endl;
      cin>>A.PReal;
      cout<<endl<<"Parte imaginaria:"<<endl;
      cin>>A.PImaginaria;
      cout<<"Numero Complejo: "<<A.PReal<<" + "<<A.PImaginaria<<"i"<<endl;
      cout<<endl<<"**Introduzca el segundo numero COMPLEJO"<<endl<<"Parte real:"<<endl;
      cin>>B.PReal;
      cout<<endl<<"Parte imaginaria:"<<endl;
      cin>>B.PImaginaria;
      cout<<"Numero Complejo: "<<B.PReal<<" + "<<B.PImaginaria<<"i"<<endl;
      // suma de C = A + B
      C.PReal = A.PReal + B.PReal;
      C.PImaginaria = A.PImaginaria + B.PImaginaria;
      // multiplicacion D = A * B
      D.PReal = A.PReal*B.PReal - A.PImaginaria*B.PImaginaria;
      D.PImaginaria = A.PReal*B.PImaginaria + A.PImaginaria*B.PReal;
      // comprar C con D (la suma y la multiplicación de A y B)
      if (C.PReal == D.PReal && C.PImaginaria == D.PImaginaria) {
            cout<<endl<< endl<<"La suma y multiplicación de los numeros complejos son
iguales:"<<endl;
      }
      else {
            cout<<endl<< endl<<"La suma y multiplicación de los numeros complejos NO son
iguales"<<endl;
      }
      cout<<"Numero Complejo A+B: "<<C.PReal<<" + "<<C.PImaginaria<<"i"<< endl;
      cout<<"Numero Complejo A*B: "<<D.PReal<<" + "<<D.PImaginaria<<"i"<< endl;
    return 0;
}
```

En el se han identificado las instrucciones que manipular directamente los datos (de color rojo) y las instrucciones de control (de color azul), es decir aquellas que no manipulan datos y que manejan el flujo, las condiciones, los ciclos, los retornos dentro del programa.

Cómo se puede observar, las instrucciones de color rojo deben tomar en cuenta la estructura de datos Complejo, su definición, el nombre y tipos de los campos, y las posibles operaciones o funciones que se les pueden aplicar a los números complejos. De la misma forma, también se puede observar que las instrucciones en azul permiten obtener interacción con la consola de entrada y/o salida, controlar las salidas condicionadas que requiere el problema.
Si se necesita resolver otros problemas como por ejemplo: obtener la sumatoria de las partes reales de un arreglo de números complejos, o bien ordenar un arreglo de complejos por la parte imaginaria, o bien obtener la productoria o sumatoria del arreglo, entre otros posibles; estas instrucciones en azul cambiarían según los requerimientos. Sin embargo, en estos problemas las instrucciones para manipular los datos, las resaltadas en rojo, serían iguales pues la definición de la estructura, la manera de tener acceso, la operación de suma, multiplicación y comparación se realizan de la misma manera sobre la estructura Complejo que se ha definido.

La idea es entonces extraer del la solución anterior las instrucciones en rojo para crear una abstracción donde cada instrucción o conjunto de instrucciones tengan una definición o significado claro para la manipulación de los números complejos. Esta extracción consiste en separar en archivos separados estas instrucciones en rojo. Lo siguiente es la definición de un archivo Complejo.h en C++ que lo contiene:

```
# ifndef COMPLEJO
# define COMPLEJO
/****Estructura Complejo
struct Complejo
 {
   double PReal;
   double PImaginaria;
 };
/****Prototipos
Complejo Llenar(double, double);
double Real(Complejo);
double Imaginaria(Complejo);
Complejo Sumar(Complejo,Complejo);
Complejo Multiplicar(Complejo,Complejo);
bool Igual(Complejo,Complejo);
/******Implementación
Complejo Llenar(double A, double B) {
       Complejo C;
       C.PReal = A;
       C.PImaginaria = B;
```

```
	return C;
}
double Real(Complejo C) {
	return C.PReal;
}
double Imaginaria(Complejo C) {
	return C.PImaginaria;
}
Complejo Sumar(Complejo A, Complejo B) {
	Complejo C;
	C.PReal = A.PReal + B.PReal;
	C.PImaginaria = A.PImaginaria + B.PImaginaria;
	return C;
}
Complejo Multiplicar(Complejo A, Complejo B) {
	Complejo C;
	C.PReal = A.PReal*B.PReal - A.PImaginaria*B.PImaginaria;
	C.PImaginaria = A.PReal*B.PImaginaria + A.PImaginaria*B.PReal;
	return C;
}
bool Igual(Complejo A, Complejo B) {
	return (A.PReal == B.PReal && A.PImaginaria == B.PImaginaria);
}
# endif
```

Utilizando este archivo se puede presentar una solución al problema original de la siguiente forma:

```
#include <iostream>
using namespace std;
#include "Complejo.h"
void imprimir(Complejo C1) {
	cout<<"Numero Complejo: "<<Real(C1)<< " + " <<Imaginaria(C1)<< "i"<<endl;
}
int main()
{
	Complejo A,B;
	double I,J;
	cout<<"** Introduzca el primer numero COMPLEJO"<<endl<<"Parte real:"<<endl;
	cin>>I;
	cout<<endl<<"Parte imaginaria:"<<endl;
	cin>>J;
	A = Llenar(I,J);
	cout<<endl<<"** Introduzca el segundo numero COMPLEJO"<<endl<<"Parte real:"<<endl;
    cin>>I;
	cout<<endl<<"Parte imaginaria:"<<endl;
```

```
        cin>>J;
        B = Llenar(I,J);
        if (Igual(Sumar(A,B),Multiplicar(A,B))) {
                cout<<"La suma y multiplicación de los numeros complejos son iguales:"<<endl;
        }
        else {
                cout<<"La suma y multiplicación de los numeros complejos NO son iguales"<<endl;
        }
        imprimir(Sumar(A,B));
        imprimir(Multiplicar(A,B));
    return 0;
}
```

Dada la última solución y comparándola con la primera piense en: características de legibilidad, claridad y portabilidad del código, reutilización de las funciones para resolver otros problemas que utilicen números complejos, nivel de abstracción de la manipulación de los números complejos, corrección de código, migración a otros lenguajes de programación, etc.

Lo que se acaba de realizar en el ejercicio anterior es lo que se conoce como un TDA (**T**ipo de **D**atos **A**bstracto) de números complejos y a continuación se presenta el marco teórico de este procedimiento o paradigma para resolver problemas, el cual es el preámbulo para orientación por objetos.

Qué es un TDA?

Según los libros:

• "Un TDA es un modelo matemático con una colección de operaciones definidas sobre el modelo" (Aho, Hoperoft y Ullman. "Fundamental Structures of Computer Science", 1981).

• "Una clase de objetos definida por una especificación independiente de la representación" (Guttag "Abstract Data Type and development of data structures " ACM . Vol 20-6, 1977)

• "Es un tipo de dato definido por el usuario a través de una especificación y una implementación de los objetos abstractos". (Rowe , "types" ACM sigplan, Vol 16-1, 1980).

Una posible explicación:

* Una calculadora es un ejemplo de un TDA que maneja objetos de cantidades numéricas y las operaciones aritméticas sobre dichas cantidades. Usa el sistema decimal para las cantidades y realiza operaciones de suma, resta, multiplicación, etc. Sin embargo, ¿ud. sabe cómo una

calculadora representa las cantidades internamente? ¿En Binario? ¿Decimal? ¿Palitos? ¿piedritas?. NO!, no lo sabe y tampoco le hace falta para usar la calculadora.

Un sistema de numeración es un ejemplo de un tipo de dato abstracto que representa el concepto de cantidad. Los siguientes son ejemplos de sistemas de numeración: Romano: I, V, X, L, C, D, M; Decimal: 0, 1, 2, 3,...., 9; Maya: . .. ... __ _._

Para algunos autores un TDA no es más que una estructura algebraica que representa una entidad y las operaciones que permitan manipularla. La característica más importante de esta representación es que intenta "**crear una protección de las entidades representadas**", es decir:

Oculta la representación e implementación de la entidad y sus operaciones. Ud. no sabe como la calculadora representa las cantidades ni como realiza la operación de sumar.

Otra característica es que los datos **sólo se manipulan a través de sus operaciones**. Ud. solo puede realizar las operaciones que la calculadora le permite realizar y nada más.

¿Cómo logramos crear dicha protección?. Según la ecuación de Wirth definimos un programa asi:

Programa = Datos + Algoritmos

Si podemos separar e identificar en un algoritmo las instrucciones que manipulan los datos de las instrucciones que indican control, entonces podemos reescribir la ecuación como:

Programa = Datos + Algoritmos-Datos + Algoritmo-Control

Donde TDA = Datos + Algoritmos-Datos, y por tanto la ecuación queda así:

Programa = TDA + Algoritmo de Control

Una definición de TDA podría se la siguiente:

> **Es un tipo de dato definido por el usuario para representar una entidad (abstracción) a través de sus características (datos o atributos) y sus operaciones o funciones (algoritmos que manipulan los datos).**
>
> **TDA = Datos + Algoritmos-Datos.**

Especificación de un TDA

Es la descripción formal del tipo de dato que representa a una entidad a través de sus propiedades y su comportamiento. Dos niveles:

1) **Especificación Sintáctica**:

¿Qué hace? Especificación de las entidades y sus propiedades (interfaz)

• Definir el nombre de las entidades abstractas.

• Definir el nombre de las operaciones indicando el dominio (argumentos) y el codominio o rango (los valores de retorno)

2) **Especificación Semántica**:

¿Cómo lo hace? Descripción de la representación del objeto (estructura de los datos) y desarrollo de las operaciones.

• Definir el significado de cada operación usando los símbolos definidos en la especificación sintáctica.

La especificación semántica puede ser de dos tipos: (1) Informal, a través del lenguaje natural y (2) Formal, rigurosa y fundamentada matemáticamente. Para la representación formal se puede usar el enfoque operacional y enfoque algebraico. La más utilizada es el enfoque operacional que usa modelos abstractos, a través de:

- Define la semántica del TDA especificando el significado de cada operación en términos de otros modelos, llamado modelo de referencia (formalmente definido).
- Denotará la semántica de cada operación, mediante la definición de un procedimiento o función indicando una precondición y postcondición definidas sobre el modelo de referencia.

{P} Pre-condición: condiciones que deben cumplirse antes de realizar la operación.

{Q} Post- condición: condiciones que se cumplen una vez realizada la operación.

La notación usual {P} S {Q} donde S es la función o procedimiento.

Ejemplo de especificación de un TDA

Ejemplo: TDA COMPLEJO cuyos objetos son los números complejos (parte real e imaginaria) X = (X1, X2). Parte imaginaria es la raíz cuadra de un número negativo.

Especificación Sintáctica:

Nombre del TDA: COMPLEJO			
Nombre de la operación	**Dominio**	**Rango (Codominio)**	**Tipo**
SUMAR	COMPLEJO x COMPLEJO	COMPLEJO	T
MULTIPLICAR	COMPLEJO x COMPLEJO	COMPLEJO	T
IGUAL	COMPLEJO x COMPLEJO	COMPLEJO	A

CREAR_COMPLEJO	Real x Real	COMPLEJO	C
PARTE_REAL	COMPLEJO	Real	A
PARTE_IMAGINARIA	COMPLEJO	Real	A

Tipos de operaciones:

- CONSTRUCTORAS: crear objetos del TDA, p.e : CREAR_COMPLEJO
- ACCESO O ANALIZADORAS: permite obtener componentes del TDA como resultado, p.e: PARTE_REAL.
- TRANSFORMACIÓN O SIMPLIFICADORAS: cambios al TDA y retornan un objeto del TDA como resultado, p.e: SUMAR.

Especificación Semántica:

función SUMAR (X,Y: COMPLEJO): COMPLEJO

Pre-Cond: X,Y son del tipo complejo

Si X = (X1,X2) y Y = (Y1,Y2) entonces:

Z = (Z1,Z2) donde Z1 = X1 + Y1 y Z2 = X2 + Y2

Post-Cond: Z es del tipo complejo, Z = X + Y

finfunción

función MULTIPLICAR (X,Y: COMPLEJO): COMPLEJO

Pre-Cond: X,Y son del tipo complejo

Si X = (X1,X2) y Y =(Y1,Y2) entonces:

Z = (Z1,Z2) tal que Z1 = X1 * Y1 – X2 * Y2

Y Z2 = X1 * Y2 + X2 * Y1

Post-Cond: Z es complejo, Z = X * Y

finfunción

función IGUAL (X,Y: COMPLEJO): Booleano

Pre-Cond: X,Y son del tipo complejo

Si X=(X1,X2) y Y=(Y1,Y2) entonces

Si X1 = Y1 y X2 = Y2 entonces Resultado = Verdadero.

Sino Resultado = Falso

Post-Cond: Resultado de comparar (X = Y)

finfunción

función PARTE_REAL (X:COMPLEJO): Real

Pre-Cond: X es del tipo complejo

Post-Cond: Si X = (X1,X2) entonces Resultado = X1

finfunción

función CREAR_COMPLEJO (X1,X2: Real): COMPLEJO

Pre-Cond: X1 y X2 son del tipo Real.

Post-Cond: X = (X1,X2) resultado X es COMPLEJO

finfunción

Ejemplo de implementación de un TDA

Elegir una representación concreta para el TDA en términos de las estructuras de datos provistas por un lenguaje de programación o pseudo lenguaje y codificar los procedimientos basándose en la estructura seleccionada.

El siguiente es un ejemplo de cómo implementar en un pseudo lenguaje el TDA COMPLEJO anteriormente especificado:

```
tipo COMPLEJO = REGISTRO
                    RE : Real;
                    IMAG: Real;
                finREGISTRO

función SUMAR (X,Y: COMPLEJO): COMPLEJO
    Z COMPLEJO;
    Z.RE = X.RE + Y.RE;
    Z.IMAG = X.IMAG + Y.IMAG;
    Retornar Z;
f.función
función PARTE_REAL (X:COMPLEJO): Real
    Retornar X.RE;
f.función
```

La implementación de las demás operaciones se deja como actividad para el lector.

A nivel del usuario que utiliza el TDA para resolver problemas sólo interesa la especificación sintáctica del TDA, pues es lo que me permite utilizarlo: crear instancias de variables del tipo y aplicar operaciones sobre esas variables.

Ejemplo de uso de un TDA

Usar el TDA permite aprovechar el nivel de abstracción en el desarrollo de un problema. Por ejemplo: Resolver el problema de verificación si la suma y multiplicación de 2 números complejos producen el mismo número complejo. Solución en pseudo lenguaje:

```
INICIO // Programa principal
        X, Y COMPLEJO
        A Booleano
        X = CREAR_COMPLEJO(3,-5)
        Y = CREAR_COMPLEJO(8,-3)
        A = VERIFICAR1(X,Y)
        Si A = verdadero entonces imprimir "Son iguales la suma y la multiplicación"
        Sino imprimir "NO son iguales la suma y la multiplicación"
        Fsi
FIN
 función VERIFICAR1 (X,Y: COMPLEJO): Booleano  // Función Verificar1
        Z1,Z2 COMPLEJO
        Z1 = SUMAR (X,Y)
        Z2 = MULTIPLICAR (X,Y)
        RETORNAR IGUAL (Z1,Z2)
 f.función
 función VERIFICAR2 (X,Y: COMPLEJO): Booleano // Función Verificar2
        RETORNAR IGUAL (SUMAR (X,Y), MULTIPLICAR (X,Y) )
 f.función
```

Se provee al lector de otra versión función VERIFICAR2 que realiza la misma operación sobre los números complejos. Note que VERIFICAR1 no es una operación del TDA, por qué?

Ventajas de uso de un TDA

- Herramienta para resolver problemas (nivel de abstracción)
- Independencia entre la forma de representar la información y la solución del problema → portabilidad de la solución.
- Favorece la especificación, verificación y depuración de programas.
- Contribuye a la flexibilidad de los cambios.