Estructura de Datos y Algoritmos

Tipos abstractos de datos

1. Introducción

Para empezar es util comparar un tipo abstracto de datos con la noción más familiar de procedimiento (función que retorna tipo void).

Los procedimientos, **generalizan** la noción de operador. En vez de estar limitado a los operadores predefinidos de un lenguaje de programación, el programador es libre de definir sus propios operadores y aplicarlos a operandos que no necesitan ser tipos básicos.

Un ejemplo es un programa de multiplicación de matrices.

Otra ventaja de definir procedimientos es que pueden ser utilizados para **encapsular** partes de un algoritmo, localizando en distintas secciones de un programa las sentencias relevantes a un cierto aspecto de un programa.

Un ejemplo de encapsulamiento es la definición de un procedimiento para leer toda la entrada y checkear su validez. La ventaja del encapsulamiento es que si decidimos por ejemplo checkear que las entradas son no negativas necesitamos cambiar pocas lineas de código.

Definición 1.1 (Tipo abstracto de datos) *Podemos pensar un tipo abstracto de datos (TAD) como un modelo matemático con una colección de operaciones definidas en el modelo.*

Por ejemplo, conjuntos de enteros junto con las operaciones de unión, intersección y diferencia de conjuntos son un ejemplo de TAD.

En un TAD, las operaciones pueden tomar como operandos no solo instancias del TAD que está siendo definido sino otros tipos de operandos, por ejemplo, enteros o instancias de otro TAD, y el resultado de una operación puede ser otro que la instancia del TAD. Sin embargo, asumimos que al menos un operando o el resultado de una operación es el TAD en cuestión.

Las dos propiedades de procedimientos mencionados anteriormente **generalización** *y* **encapsulamiento** *se aplican a TADs.*

TADs son generalizaciones de tipos primitivos de datos (int, double, char) del mismo modo que procedimientos son generalizaciones de operaciones primitivas (+,-,).*

TADs encapsulan tipos de datos en el sentido de que la definición del tipo y las operaciones en ese tipo se localizan en una sección del programa.

Si deseamos cambiar la implementación de un TAD, sabemos donde mirar. Más aún, fuera de la sección en la cual las operaciones del TAD están definidas, podemos tratar al TAD como si fuera un tipo primitivo, no miramos la implementación subyacente.

Debemos enfatizar que no hay limite en el número de operaciones que se pueden aplicar a instancias de un modelo matemático particular.

Ejemplo 1.2 (Operaciones en Set) *Algunas de las operaciones que se pueden aplicar al TAD SET son:*

1. *Makenull(A). Inicializa el conjunto A al conjunto vacio.*

2. *Add_elem(A,a). Agrega el elemento a al conjunto A.*

3. *Union(A,B,C). Toma dos sets A y B y asigna la unión de A y B al conjunto C.*

4. *Size(A). Toma un conjunto A y devuelve un entero cuyo valor es la cantidad de elementos de A.*

Definición 1.3 (Implementación de un TAD) *La implementación de un TAD es la traducción en sentencias de un lenguaje de programación de la declaración que define el tipo, de las declaraciones que definen una variable como del tipo más los procedimientos que definen cada operación del tipo de datos.*

La implementación elige una estructura de datos *para representar el TAD. Cada estructura de datos se construye a partir de los tipos de datos básicos del lenguaje de programación en cuestión. Por ejemplo si consideramos conjuntos finitos, podríamos implementar el TAD SET de elementos de tipo A como un array de elementos de tipo A con tope declarado como sigue:*

```
#define MAX 5000

struct SET
{A e[MAX];
 int tope;
};
```

Una razón importante para definir dos TADs como distintos si tienen el mismo modelo matemático pero distintas operaciones es que lo apropiado de una implementación depende mucho de las operaciones que se realizan.

Cuando hablamos de estructuras de datos nos referimos a tipos definidos por el usuario (no predefinidos como int, float, etc) sino como son los arreglos y las estructuras.

Veamos el rol de los tipos abstractos de datos en el proceso de diseño de programas.

Abstracción: Consiste en ignorar los detalles de la manera particular en que está hecha una cosa quedandonos solamente con su visión general.

Nos interesa extender la idea de abstracción a los tipos de datos.

Un tipo abstracto de datos es un modelo matemático compuesto por una colección de operaciones definidas sobre el modelo y un nombre que lo identifica.

Se define como un conjunto de valores que pueden tomar los datos del tipo junto a las operaciones que los manipulan.

En C, las operaciones son subprogramas (procedimientos, funciones).

Ejemplo de tipo abstracto de datos:

Conjunto de números enteros con las operaciones de unión, intersección y diferencia.

Se separa la especificación del tipo (que hace) de la implementación (como lo hace).

El tipo concreto provisto para definir el TAD en una implementación dada se llama **representación** del TAD.

1.1. Uso de Tipos Abstractos en C/C++

Problema: Quiero escribir un programa que simule una calculadora con la que se pueda operar con números racionales dados en la forma m/n con m,n enteros.

Se podrá sumar, restar, multiplicar o dividir un número con otro lo cual da lugar a un resultado.

Necesitaremos una variable para almacenar el último resultado computado (simula el visor de la calculadora) y otra para almacenar cada operando a ser combinado con el visor por medio de las operaciones aritméticas.

Los racionales no son un tipo de datos primitivo, ahora bien, queremos continuar el diseño del programa sin detenernos a pensar como los representaremos.

La idea es **Racional** es un tipo abstracto de datos que será refinado posteriormente.

Escribiremos entonces un módulo de definición:

```
#ifndef Racionales
#define Racionales
typedef struct TipoRacional* Racional;

< declaraciones de operaciones sobre Racionales >

#endif Racionales
```

La declaración typedef introduce el nombre del TAD. Luego iremos agregando especificaciones de las operaciones que sean necesarias.

Notese que no definimos el tipo, en ese caso decimos el tipo Racional es opaco. Es decir: en el módulo de definición solo **especificamos** el tipo, no lo implementamos.

La ventaja de esto es que nos permite trabajar a un nivel de abstracción en particular posponer el problema de implementar el tipo.

El programa principal no dependerá de la implementación particular que se elija sino solo de la especificación del tipo, por lo tanto, modificar la implementación del tipo no implica tener que modificar el programa principal.

El programa principal incluirá el módulo Racionales.

Consideremos los **comandos** aceptados por el programa. Introducimos un tipo enumerado con las operaciones:

```
typedef enum {suma, resta, producto, division, borrar, ingreso, fin} Comando;
```

declaramos una variable del tipo para almacenar cada comando ingresado:

```
Comando c;
```

El programa principal tendrá la forma:

```
Racional visor, opnd;

do{
        Leercomando(c);
        switch (c){
```

```
                case suma: <proceso suma>... break;
                case resta: <proceso resta>... break;
                case producto: <proceso producto>... break;
                case division: <proceso division>... break;
                case borrar: <proceso borrar>... break;
                case ingreso: <proceso ingreso>... break;
                }
        <mostrar visor si corresponde>
while (c!=fin);
```

Al refinar los procesos correspondientes a los comandos obtenemos las operaciones requeridas para el tipo Racional, por ejemplo:

```
<proceso suma>
LeerRacional (opnd);
SumaRacional(opnd, visor);
<...>
```

Las otras operaciones funcionan de manera similar.

Agregaremos operaciones básicas que pueden ser utilizadas por las operaciones anteriores:

```
Racional CrearRacional(int m, int n);
/* precondicion n<>0, retorna m/n */

int Numerador (Racional q);
/* precondicion: q es valido, retorna el numerador */

int Denominador (Racional q);
/* precondicion: q es valido, retorna el denominador */

bool EsValido (Racional q);
/* q no fue borrado */
```

En este punto, el programa principal y el módulo racionales pueden desarrollarse en forma separada. La comunicación entre ambos ha quedado establecida. Si se compila el modulo de especificación de racionales el programa principal puede ser compilado también pues solo depende del módulo de especificación.

En forma independiente puede desarrollarse y compilarse el módulo de **implementación** de los racionales. Esto favorese el trabajo en grupo que puede operar en paralelo una vez definidos los módulos de especificación necesarios.

Veamos ahora como se puede refinar el TAD Racional. Debemos escribir el módulo de implementación Racionales

```
#include Racionales.h"

struct TipoRacional {
    <representacion elegida>;
```

```
}
< implementaciones de procedimientos >
```

Finalmente mostramos el visor luego de procesado cada comando excepto el de finalización:

```
< mostrar visor si corresponde >
      if comando != fin
      EscribirRacional(visor)
<...>
```

Clasificamos las operaciones basicas anteriores del siguiente modo:

1. Operaciones Constructoras: forman racionales a partir del numerador y denominador, ejemplo CrearRacional.
2. Operaciones Destructoras o Selectoras: Numerador y Denominador, permiten obtener los componentes del tipo.
3. Operaciones Observadoras: EsVálido, permite .^{o}bservar.^{el} estado del elemento (si es valido o fue borrado).

Los tipos abstractos que se introducen sólo con su nombre en el módulo de definición se denominan **OPACOS**.

En la implementación **deben** ser definidos como **punteros** a variables que tendrán como tipo la presentación que se elija dar al tipo abstracto. Esto NO limita las posibilidades de representación. Trabajamos con punteros a las representaciones en lugar de con las representaciones mismas.

En nuestro caso podemos tener:

```
struct TipoRacional{
      int num,denom;
}
```

en cuyo caso el estado "borrado"puede ser representado como el puntero NULL.

Un ejemplo de implementación es la suma de racionales que definimos como sigue:

```
void SumaRacional(Racional q, Racional r)
{
    unsigned int numq, denomq, numr, denomr;
    if (EsValido(r)){
      numq=Numerador(q);
      denomq=Denominador(q);
      numr=Numerador(r);
      denomr=Denominador(r);
      r- >num = numq*denomr + denomq*numr;
      r- >denom = denomq*denomr;
      Normalizar(r);
```

```
    }
}
```

El procedimiento **Normalizar** calcularía la forma simplificada del racional dado como parametro. Una posible forma de definirla es tomar los restos de dividir numerador y denominador entre los números 2 a minimo del numerador y denominador. Cuando los restos dan ambos 0 se dividen entre el número.

Notar que SumaRacional podría definirse sin acceder a la representación haciendo uso de la operación **CrearRacional**. Esto tiene la ventaja de que se puede definir sin conocer la representación del TAD.

Conclusiones

- Los programas que usan tipos abstractos permanecen independientes de la implementación de estos.
- Las representaciones son opacas para los programas lo cual garantiza:
    1. que los programas manipulen los objetos del tipo abstracto solo a través de las operaciones definidas
    2. que los programas no deben modificarse si la implementación del tipo abstracto se modifica.
- Se tiene un mecanismo de módulos que favorece el trabajo en paralelo.

Repaso

- La **especificación** de un TAD se escribe en un módulo de definición. El nombre del tipo se declara sin que sea definido (tipo opaco). Las operaciones se especifican como prototipos indicando precondición y postcondición.
- La **implementación** del TAD se escribe en el correspondiente módulo de implementación. La representación se da en la definición del tipo y luego se escriben las correspondientes implementaciones de procedimientos.