Vol. (6) 2: pp. 56-62

IMPLEMENTACIÓN DE UN TIPO ABSTRACTO DE DATOS PARA GESTIONAR CONJUNTOS USANDO EL LENGUAJE DE PROGRAMACIÓN C++

** Edgar Ruiz L.*
** Hilmar Hinojosa L.*

RESUMEN

El artículo presenta la implementación de un tipo abstracto de datos para representar el concepto matemático de la teoría de conjuntos. El programa ha sido escrito en lenguaje de programación C++ aplicando el paradigma de la programación orientada a objetos mediante el compilador Dev C++ v. 4.1; un compilador GNU con licencia GPL.

Palabras Claves: Tipo abstracto de datos. Teoría de conjuntos. Estructura algebraica. Sobrecarga de operadores.

ABSTRACT

This article presents the implementation of a data abstract type to represent the Set Theory Mathematical Concept, The program has been written in C++ Program Language applying the Object Oriented Programming Paradigm through a Dev C++ v.4.1 Compiler, a GNU compiler with GPL licence.

Key Words: Data abstract type. Set theory. Algebraic structure. Operators overcharge.

INTRODUCCIÓN

El concepto matemático de conjuntos se encuentra implementado como un tipo de dato primitivo en lenguajes como Pascal, mediante la cláusula *set of* pueden declararse variables del tipo conjunto. En el lenguaje C++ no existe un equivalente al *set of* de Pascal, por ello el programador debe implementarlo de algún modo.

Como acaba de verse un *Set* o Conjunto, no esta incorporado como una facilidad primitiva del lenguaje C++, por ello; cuando se quiere aplicar estos conceptos, el programador debe hacerlo creando su propio tipo abstracto de datos *set*. Para ello se crea una clase llamada conjunto que almacena datos de tipo entero y a partir de ella poder gestionar las operaciones propias de todo conjunto como son: la unión, intersección, pertenencia, diferencia, comparación y cardinalidad.

OBJETIVO

El objetivo del presente trabajo es definir una clase que gestione conjuntos de enteros. Las operaciones para el conjunto de enteros son las siguientes:

- **Unión.-** La unión de dos conjuntos A y B, denotado por A ▾ B ; es el conjunto de todos los elementos que están en A, en B o en ambos.
- **Intersección.-** La intersección de dos conjuntos A y B, denotado por A ▾ B; es el conjunto de todos los elementos que pertenecen a ambos A y B simultáneamente.
- **Diferencia.-** La diferencia de conjuntos A y B, denotado por A - B ; es el conjunto de elementos que pertenecen a A pero no pertenecen a B
- **Pertenencia.-** El número entero x es un miembro o pertenece al conjunto A si x es un elemento de A. Esta operación se denota por x ▾ A .
- **Comparaciones.-** Los conjuntos pueden compararse entre si mediante el uso de operadores relacionales.
  - **a.** Subconjunto.- El conjunto A es un subconjunto del conjunto B si y solo si todo elemento de A es también un elemento de B.
  - **b.** Igualdad.- El conjunto A es igual al conjunto B si y solo si cada elemento de A está en B y cada elemento de B está en A.
  - **c.** Desigualdad.- El conjunto A es diferente del conjunto B si y solo si A y B no son iguales.
- **Cardinalidad.-** La cardinalidad de A, denotado por #A es el número de elementos de A

* Instituto de Investigación
Facultad de Ingeniería Industrial, UNMSM
E-mail: iifi@unmsm.edu.pe

A decir verdad, los compiladores modernos de C++ incorporan en la Biblioteca Estándar de Plantillas conocida como STL(Standard Template Library) un grupo de 60 algoritmos muy útiles para los programadores; entre ellos, están los que corresponden a las operaciones de conjunto y multiconjuntos sobre secuencias de datos como son: *includes( ), set_union( ), set_interseccion( ), setdifference( ),y set_symetric_difference( )*, los cuales se pueden aplicar siempre y cuando la secuencia de datos de entrada este ordenada de lo contrario su comportamiento es ineficiente y no corresponde a la teoría de conjuntos. Sin embargo, nada impide que un programador pueda plantear soluciones a su medida, de allí que el artículo presenta una implementación particular de conjuntos.

Para ver un conjunto desde un punto de vista matemático se debe responder a las siguientes cuestiones:

1. ¿Qué tipos de datos se requieren para, determinar un conjunto?
2. ¿Qué tipos de operaciones están definidas para estos datos?

TIPO ABSTRACTO DE DATOS

A la descripción completa de ambos: los datos y las operaciones requeridas para ellos se le llama Tipo Abstracto de Datos (Abstract Data Type). Un TAD en ciencia de la computación es un término basado en un concepto matemático llamado estructura algebraica. Una estructura algebraica, consiste del dominio de objetos y las operaciones que se pueden realizar con esos objetos. Puede decirse que hasta aquí se ha llegado a nivel de diseño. El proceso que sigue consiste en definir el nuevo tipo en términos de código en algún lenguaje de programación. Resumiendo; en la construcción de un TAD, son necesarios dos pasos: primero, definir él o los tipos de datos, y segundo implementar dicho TAD como una clase.

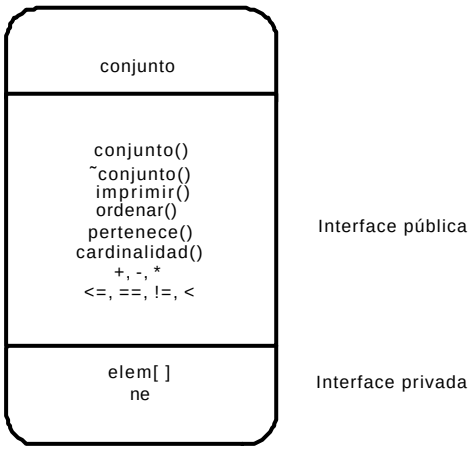

Figura 1. El Tipo Abstracto de Datos conjunto

La construcción se realiza con el lenguaje de programación C++, un lenguaje orientado a objetos. Para la definición del tipo de datos se han escogido los enteros y para la implementación se plantea la clase conjunto.

IMPLEMENTACIÓN DEL TAD CONJUNTO

La implementación del TAD conjunto se hace mediante una clase a la que se llama conjunto.

El diseño de la clase conjunto se presenta en la Figura 1.

El TAD conjunto se implementa en un archivo de cabecera o *header* llamado **conjunto.h**; y el archivo de prueba para la clase conjunto se escribe en el archivo fuente **conjun1.cpp**. A esto se le conoce como compilación separada. Los listados de ambos archivos se presentan en las Figuras 2 y 3 respectivamente.

Nótese que para las operaciones del TAD conjunto se han sobrecargado los operadores del modo siguiente:

+	para la unión
-	para la diferencia
*	para la intersección
<	para el subconjunto
==	para la igualdad
¡=	para la diferencia y
<=	para el subconjunto propio

La sobrecarga de operadores es una facilidad muy poderosa soportada por C++, que permite al usuario añadir un nuevo comportamiento a los operadores primitivos proporcionados por el lenguaje. Explicamos brevemente este concepto: sabemos que el operador + permite sumar dos objetos del mismo tipo como por ejemplo; dos enteros o dos reales, sin embargo podemos hacer que este mismo operador + adquiera dos nuevos significados útiles para nuestro TAD conjunto según lo siguiente:

(1) Añadir o incluir un nuevo elemento al conjunto y
(2) Realizar la unión de conjuntos.

Las secciones de código en C++, para estas operaciones se presentan enseguida.

```
//implementacion de la libreria conjunto
#ifndef  CONJUNTO_H
#define  CONJUNTO_H
//Autores: Edgar Ruiz L. y Hilmar Hinojosa L.

const int N = 100;
//declarando la clase conjunto
class conjunto{
public:
     conjunto();  //constructor
     ~conjunto(); //destuctor
    void imprimir();
     void ordenar();
     bool pertenece(int x);
    int cardinalidad();
     void operator+(int x);
    //funciones amigas
     friend conjunto operator+(conjunto c1 , conjunto c2);
     friend conjunto operator*(conjunto c1 , conjunto c2);
     friend conjunto operator-(conjunto c1 , conjunto c2);
     //comparaciones
     friend bool operator<=(conjunto c1, conjunto c2);
     friend bool operator==(conjunto c1, conjunto c2);
     friend bool operator!=(conjunto c1, conjunto c2);
     friend bool operator<(conjunto c1, conjunto c2);
private:
     int elem[N];
    int ne;
};
//constructor
conjunto::conjunto()
{
    int i;
     for(i=0;i<N;i++)
        elem[i] = 0;
    ne = 0;
}
//destuctor
conjunto::~conjunto()
{
}
//sobrecarga para el operador + para insertar un nuevo elemento
void conjunto::operator+(int x )
{
     if ( pertenece(x) == false )
    {  elem[ne] = x;
       ne++;
    }
    else
        cout<<"Elemento ya existe"<<endl;
}
//imprimir los elementos del conjunto
void conjunto::imprimir()
{
    int i;
     if ( ne > 0 )
     {   ordenar();
        for(i=0;i<ne;i++)
           cout<<elem[i]<<"\t";
        cout<<endl;
    }else
        cout<<"El conjunto esta vacio"<<endl;
}

//ordenar los elementos del conjunto
void  conjunto::ordenar()
{
    int i,j,x;
     if ( ne > 1 )
     {  for(i=0;i<ne-1;i++)
          for(j=i+1;j<ne;j++)
             if ( elem[i] > elem[j] )
             {   x = elem[i];
                elem[i] = elem[j];
                elem[j] = x;
             }
    }
}
// el elemento pertenece al conjunto?
bool conjunto::pertenece(int x)
{
    int i;
    bool b;
     b = false;
    i = 0;
     while( b == false && i < ne)
        if (elem[i] == x)
           b = true;
        else
           i++;
    return b;
}
//cardinalidad: numero de elementos del conjunto
int  conjunto::cardinalidad()
{
    return ne;
}
//definiendo la union de conjuntos
```

Figura 2a. Librería del *conjunto.h*

```
conjunto operator+(conjunto c1, conjunto c2)
{
    conjunto c3;
   int x, i;
    for(i=0;i<c1.ne;i++)
       c3.elem[i] = c1.elem[i];
    c3.ne = c1.ne;
    for(i=0;i<c2.ne;i++)
    {   x = c2.elem[i];
        if (c3.pertenece(x) == false)
       {   c3.elem[c3.ne] = x;
          c3.ne++;
       }
    }
    return c3;
}
//definiendo la interseccion de conjuntos
conjunto operator*(conjunto c1, conjunto c2)
{
    conjunto c3;
   int x,i;
    for(i=0;i<c1.ne;i++)
    {   x = c1.elem[i];
        if ( c2.pertenece(x) == true )
       {   c3.elem[c3.ne] = x;
          c3.ne++;
       }
    }
    return c3;
}
//definiendo la diferencia de conjuntos
conjunto operator-(conjunto c1, conjunto c2)
{
    conjunto c3;
   int x,i;
    for(i=0;i<c1.ne;i++)
    {   x = c1.elem[i];
        if ( c2.pertenece(x) == false )
       {   c3.elem[c3.ne] = x;
          c3.ne++;
       }
    }
    return c3;
}
//definiendo el subconjunto propio
bool operator<=(conjunto c1, conjunto c2)
{
   int x,i;
    for(i=0;i<c1.ne;i++)
    {   x = c1.elem[i];
        if ( c2.pertenece(x) == false )
          return false;
    }
    return true;
}
//definiendo la igualdad de conjuntos
bool operator==( conjunto c1, conjunto c2 )
{
    if ( ( c1 <= c2 ) && ( c2 <= c1 ) )
       return true;
    else
       return false;
}
//definiendo la desigualdad de conjuntos
bool operator!=( conjunto c1, conjunto c2 )
{
    return !(c1 == c2);
}
//definiendo el subconjunto
bool operator<(conjunto c1, conjunto c2)
{
    if ((c1 <= c2) && (c1 != c2))
       return true;
    else
       return false;
}
#endif //fin de la libreria conjunto.h
```

Figura 2b. Librería del *conjunto.h*

```
// Archivo: conjun1.cpp
#include <iostream.h>
#include <stdlib.h>
#include "conjunto.h"

void main()   //conjun1.cpp
{
   conjunto A,B,C,D,E;
   int i,x,n;
    cout<<"Cuantos elementos desea ingresar para el Conjunto
    A ? ";
   cin>>n;
    for(i=1;i<=n;i++)
    {   cout<<"Ingrese elemento "<<i<<": ";
      cin>>x;
      A + x;
   }
    A.imprimir();
    cout<<"\nCuantos elementos desea ingresar para el Conjun-
    to B ? ";
   cin>>n;
   for(i=1;i<=n;i++)
   {   cout<<"Ingrese elemento "<<i<<": ";
      cin>>x;
      B + x;
   }
   B.imprimir();
   C = A + B;
    cout<<endl<<"\nC = A UNION B"<<endl;
   C.imprimir();
   cout<<"La cardinalidad de C es "<<C.cardinalidad()<<endl;
   D = A * B;
    cout<<endl<<"\nD = A INTERSECCION B"<<endl;
   D.imprimir();
   cout<<"La cardinalidad de D es "<<D.cardinalidad()<<endl;
   E = A - B;
    cout<<endl<<"\nE = DIFERENCIA de A y B"<<endl;
   E.imprimir();
   cout<<"La cardinalidad de E es "<<E.cardinalidad()<<endl;
   if (A <= B)
   {   cout<<"\nA es subconjunto de B"<<endl;
      if (A < B)
         cout<<"\nA es subconjunto propio de B"<<endl;
   }
   else
      cout<<"\nA no es subconjunto de B"<<endl;
   if (A == B)
      cout<<"\nA y B son iguales"<<endl;
   else
      cout<<"\nA y B son diferentes"<<endl;
   cout<<endl;
    system("PAUSE");
}
```

Figura 3. Programa del *conjun1.h*

(1) Añadir o incluir un nuevo elemento al conjunto con el operador +

```
//sobrecarga para el operador + para insertar un nuevo
elemento
void conjunto::operator+(int x)
{
    if (pertenece(x) == false)
    {  elem[ne] = x;
       ne++;
    }
    else
       cout<<"Elemento ya existe"<<endl;
}
```

(2) Realizar la unión de conjuntos con el operador +

```
//definiendo la union de conjuntos
conjunto operator+(conjunto c1, conjunto c2)
{
   conjunto c3;
   int x, i;
   for(i=0;i<c1.ne;i++)
     c3.elem[i] = c1.elem[i];
   c3.ne = c1.ne;
   for(i=0;i<c2.ne;i++)
   {  x = c2.elem[i];
      if (c3.pertenece(x) == false)
      {  c3.elem[c3.ne] = x;
         c3.ne++;
       }
   }
   return c3;
}
```

Las Figuras 4 y 5 muestran dos corridas del programa.

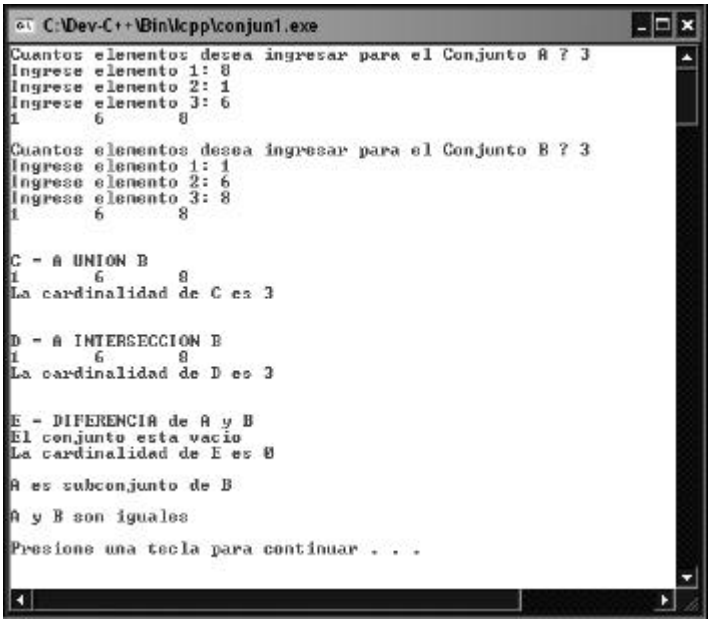

Figura 4. Una salida para el programa

Figura 5. Una segunda corrida para el programa

CONCLUSIONES

El artículo muestra que el programador puede crear tipos abstractos de datos de acuerdo a sus necesidades particulares. Esto es posible debido a que C++ es un lenguaje multiparadigma y permite aplicar el concepto de clase proporcionado por el paradigma orientado a objetos.

El programa comprueba que el concepto de sobrecarga de operadores, es una poderosa herramienta que permite a los programadores añadir funcionalidad y nuevos significado a los operadores primitivos definidos en el lenguaje C++.

A fin de que el TAD conjunto presentado sea genérico; es decir, que además de los enteros soporte flotantes, dobles, caracteres, cadenas y otros objetos el programa debe replantearse o rescribirse bajo el concepto de las plantillas o templates, que es otra de las facilidades proporcionadas por el lenguaje de programación C++, y que pueden ser materia de un articulo próximo.

BIBLIOGRAFÍA

1. **Joyanes Aguilar, L. (1997).** Turbo/Borland Pascal 7. 1ra. Ed. McGraw-Hill/Interamericana de España S.A., Madrid, España.

2. **Gamma, Erich; Helm, Richard & Others. (1999).** Introduction to Oriented Design in C++. 1ra. Ed. Addison Wesley Publishing Company, Inc. U.S.A.

3. **Stroustrup, Bjarne. (2002).** El Lenguaje de Programación C++. Edición especial. Addison Wesley - PEARSON EDUCACION S.A. España.