3. Abstracción de datos

3.1 Concepto de Abstracción y documentación

La *abstracción* consiste en descartar detalles para una mejor comprensión del elemento. Cuando hablamos en programación de abstracción, por ejemplo un programa, función, o módulo, nos importa más acerca de qué hace, frente a cómo lo hace. Así por lo tanto el programa, función o módulo lo vemos como una caja negra en la que se introduce información (entradas) y aparece una nueva información (salidas). El procesamiento que ocurre entre la información de entrada y la información de salida no es visible.

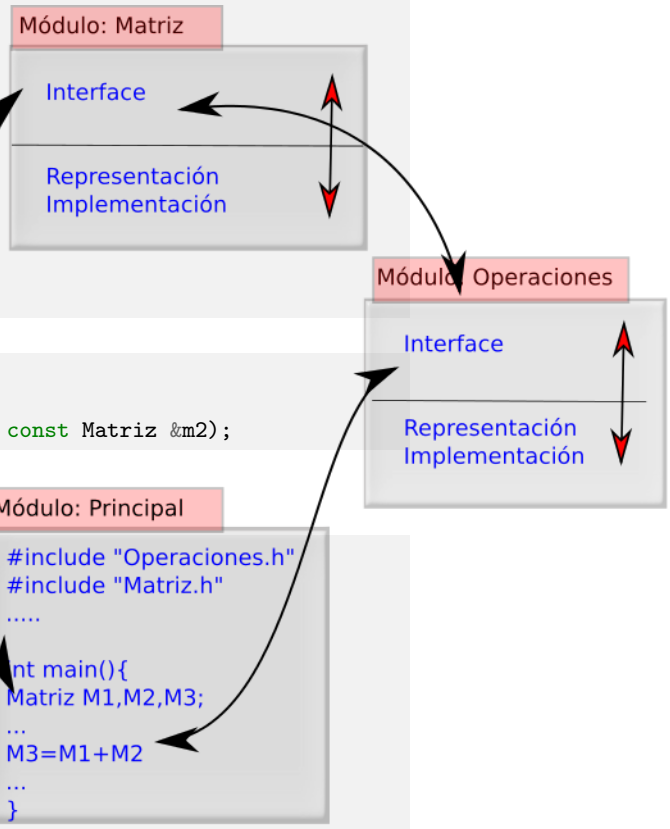
En este tema vamos a tratar acerca de los tipos de datos abstractos, T.D.A (p.e algunos probablemente ya conocidos por el lector como matriz, vector dinámico, etc), son nuevos tipos de datos con un grupo de operaciones que proporcionan la única manera de manejarlos. Por lo tanto el usuario de un T.D.A debe conocer que operaciones puede realizar sobre él pero no necesita saber nada acerca de: 1) la forma en como se almacenan los datos, 2) ni como se implementan las operaciones. De esta forma el creador del T.D.A ofrece una parte pública a los usuarios del tipo y tambien contiene una parte privada. Esta parte privada es la que caracteriza el proceso de *ocultación de información* del tipo. Un módulo se divide en dos partes: la *interfaz* que es pública; y la *implementación* y la *representación* del tipo de dato abstracto que es privada. Si un módulo necesita de otro se comunica con éste a través de su interfaz.

Ejemplo 3.1.1

```
//Modulo:Matriz
//Matriz.h
class Matriz
{
  private:
   //Representacion
   //del tipo Matriz

  public:
   Matriz();
   Matriz(Matriz &m);
   int Filas() const;
   //etc
}
```

```
//Modulo:Operaciones
//Operaciones.h
Matriz operator+ (const Matriz &m1, const Matriz &m2);
```

```
//Modulo:Principal
//principal.cpp

#include <iostream>
#include "Operaciones.h"
#include "Matriz.h"


int main ()
{
      Matriz M1, M2, M3;
      cin >> M1;
      cin >> M2;
      M3 = M1+M2;
}
```



El módulo Matriz se compone de los ficheros Matriz.cpp y Matriz.h. En Matriz.h tenemos la cabecera de las métodos de la clase Matriz junto con la representación de matriz (datos). La representación se considera oculta y por ello se califica como privada. En Matriz.cpp tenemos la implementación de los métodos de la clase Matriz, esto se considera también oculto ya que cualquier módulo ajeno a Matriz no tiene por qué saber la implementación de las funciones. El módulo Operaciones también se compone de dos ficheros: Operaciones.h y Operaciones.cpp. En nuestro ejemplo en Operaciones.h tenemos la cabecera del operador +. Esta función obtiene la suma de dos matrices en un nueva matriz.

Por lo tanto para poder implementar este operador, que se hará en el fichero Operaciones.cpp, necesita llamar a los métodos de la clase Matriz (por ejemplo para saber el numero de filas o el numero de columnas). Para ello, el módulo Operaciones se comunicará con la interfaz de Matriz para solicitar lo que necesite. Igualmente en el main, implementado en el fichero Principal.cpp necesita de Matriz y Operaciones, para ello usará la interfaz que ambos módulos le ofrecen.

□

Otro aspecto relevante de nuestro módulo es la *Documentación*. En la *Documentación* del módulo escribiremos en lenguaje natural qué hace cada método de nuestra clase/módulo. Por lo tanto cualquier usuario de nuestro módulo pueda entender que parámetros necesita el método, que resultados obtiene, cuales son las condiciones para una perfecta ejecución, etc. Para documentar existen diferentes herramientas.Una herramienta pública para documentar código es **Doxygen** `http://www.doxygen.org`

3.1.1 Abstracción por especificación

En la abstracción por especificación, se dan los detalles del módulo independientemente de la implementación. Para ello debemos dar información sobre:

¿QUÉ? SI IMPORTA ¿CÓMO? NO IMPORTA

1. **Características sintácticas**: indicar la cabecera de la función: 1)Nombre de la función,2) Parámetros de entrada junto con sus tipos, 3) Tipos de los resultados.
2. **Características semánticas**: indicar qué hace y qué devuelve la función con lenguaje natural. Por ejemplo cuando especificamos una función debemos recordar que qué hace la función si importa mientras que cómo lo hace no importa.

Ejemplo 3.1.2

Supongamos que queremos especificar el operador + de dos matrices. Si nos fijamos en los siguientes comentarios uno realiza una especificación correcta mientras que el otro no.

1. COMENTARIO 1: recorre la matriz mediante un for para las filas y otro for para las columnas y suma el elemento de la primera matriz con el de la segunda
2. COMENTARIO 2: obtiene la matriz suma de las dos matrices de entrada.

La opción válida aquí es la segunda.

□

Ejemplo 3.1.3

En este ejemplo se muestra para la función *Busqueda* una posible documentación en formato doxygen.

```
1 //busqueda.h
2
3 /**
4 * @ file ref_abst.h
5 * @ brief modulo de busqueda sobre un vector
6 **/
7 /**
```

```
8  * @ brief Busca en un vector un elemento dado
9  * @ param v: vector que contiene donde buscar el elemento de entrada \e x
10 * @ param n: numero de elementos del vector v
11 * @ param x: elemento a buscar en v
12 * @ pre
13 * - n \> 0
14 * - v tiene al menos n elementos
15 * @ return devuelve la posicion del elemento x en v
16 * @ exception devuelve -1 si x no esta en el vector
17 **/
18 int Busqueda (int *v, int n, int x);
```

Las condiciones que se deben cumplir tras ejecutar la condicion se expresan con @post

□

3.1.2 Tipo de dato abstracto

Un ***tipo de dato abstracto, T.D.A,*** es un conjunto de datos junto con unas operaciones proporcionando sobre estos una especificación que es independiente de la implementación. El conjunto de operaciones que se definen tienen que ser suficientes para que cualquier usuario del T.D.A pueda interactuar con él. Además este conjunto de operaciones debe ser minimal, esto significa que si una operación, se puede implementar en base a otras, del conjunto minimal, entonces esta nueva operación no debe estar en el conjunto minimal.

Ejemplo 3.1.4
T.D.A **Matriz**. **Especificación**:Almacena un conjunto de datos dispuestos en una serie de filas y columnas. Cada filas contiene el mismo numero de columnas.

Las operaciones son:

- Constructores (por defecto, copia, por parámetros...): construye objetos de tipo matriz
- Operadores (asignación, suma...): opera sobre una o mas matrices.
- Operadores de consulta (GetFilas, GetColumnas...): consulta los datos relativos al objeto de tipo matriz
- Operadores de modificación (SetFilas, SetColumnas...): modifica los datos del objeto matriz
- Operadores de E/S: lee o escribe una matriz sobre un flujo de entrada o salida, respectivamente.

Con respecto a las operaciones descritas, es posible que algunas no las consideremos dentro del conjunto de operaciones del T.D.A. Por ejemplo la función suma podríamos no considerarla como método del T.D.A de matriz ya que con las otras operaciones podríamos implementarla sin tener que acceder a la representación de la matriz.

□

Debemos resaltar que sobre la implementación no se dice nada en la fase de especificación. Para la implementación debemos seguir los siguientes pasos:

1. Elegir una representación.
2. Basándonos en la representación escogida, implementar las operaciones.

El T.D.A se establece con su especificación y por otro lado existe el tipo de dato dado por la representación (que datos vamos a usar para representar lo que hemos dicho en la especificación). Así cuando definimos el T.D.A debemos de establecer:

1. T.D.A. dado en el especificación.
2. **Tipo rep**: tipo a usar para representar el T.D.A. dado en la especificación, está asociado con la implementación. Según el tipo rep escogido se hará la implementación de las operaciones asociadas al T.D.A.

Ejemplo 3.1.5
T.D.A. **Fecha**

1. **Especificación**: representa una fecha en el calendario occidental "d/m/a", siendo d el día, m el mes, y a el año.
   Operaciones:
   - Constructores: constructor por defecto, constructor con una fecha determinada.
   - Consulta: acceder al dia, mes y año.
   - Modificadores: del día, mes y año
   - Escritura y Lectura de una fecha por un flujo de entrada y salida, respectivamente.
2. **Tipo rep**:
   a) Posibilidad 1:

```
class Fecha {
      private:
            int d, m, a;
```

   b) Posibilidad 2:

```
class Fecha {
      private:
            string f;
```

   c) Posibilidad 3:

```
enum Mes = {ENE, FEB, MAR, ABR, MAY, JUN, JUL, AGO, SEP, OCT, NOV, DIC}
class Fecha {
      private:
            int d, a;
            Mes m;
```

□

Función de abstracción

Es una función que transforma el ***tipo rep*** escogido con el T.D.A. dado en la especificación. Hay que tener en cuenta que el ***tipo rep*** escogido puede contener muchos más datos que los que se usan en la especificación del T.D.A., Es decir el dominio del ***tipo rep*** es un superconjunto de conjunto definido en la especificación del T.D.A.. Por lo tanto la función de abstracción tiene como objetivo establecer que datos, definidos por el ***tipo rep***, son los usados para expresar el T.D.A definido en la especificación.

$f_A : rep \longrightarrow$ T.D.A. especificación

Ejemplo 3.1.6
T.D.A **Fecha**
Tras haber elegido la posibilidad 1, en el ejemplo 3.1.5, nuestra función de abstracción se define como:

$$f_A(r) = \text{“r.d/r.m/r.a”}$$

Donde r es una instancia del objeto abstracto de tipo rep que sirve para representar el T.D.A.

□

Ejemplo 3.1.7
T.D.A. **Racional**

1. **Especificación**: representa a los números racionales de tal forma que, si n es el numerador y d es el denominador, el racional asociado es $\frac{n}{d}$
   Operaciones:
   - Constructores: constructor por defecto, constructor con unos valores concretos.
   - Consulta: acceder al numerador, denominador
   - Modificadores: del numerador, denominador
   - Escritura y Lectura de una racional por un flujo de entrada y salida, respectivamente.
2. **Tipo rep**:

```
1 class Racional {
2       private:
3             int num, dem;
```

3. **Función de abstracción**:

$$f_A(r) = \frac{r.num}{r.dem}$$

□

Invariante de la representación
Son las condiciones que debe cumplir el ***tipo rep*** para representar el T.D.A. dado en la especificación.

Ejemplo 3.1.8
T.D.A. **Racional**
El invariante de representación para un racional es que el denominador debe ser distinto de cero. Así siguiendo el ejemplo 3.1.7 diremos que siendo r un objeto de tipo rep $r.dem \neq 0$.

□

Ejemplo 3.1.9
T.D.A **Fecha**
En el ejemplo 3.1.5 habiendo escogido la posibilidad 1, el invariante de la representación contiene la siguientes condiciones:

1. $1 \leq d \leq 31$
2. $1 \leq m \leq 12$
3. Si $m = 4,6,9,11$ y $bisiesto(a) \rightarrow m \leq 30$
4. Si $m = 2$ y $bisiesto(a) \rightarrow d \leq 29$
5. Si $m = 2$ y $\neg bisiesto(a) \rightarrow d \leq 28$

□

Ejemplos completos

Ejemplo 3.1.10
T.D.A. **Polinomio**

1. **Especificación**: sucesión de reales $a_0, a_1, \cdots, a_n$ que representan polinomios con coeficientes reales del tipo $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$
   Operaciones:
   - Constructores: constructor por defecto, constructor con unos valores concretos.
   - Consulta: acceder al coeficiente del monomio i-th, consultar el grado del polinomio.
   - Modificadores: modificar el coeficiente del monomio i-th
   - Escritura y Lectura de un polinomio por un flujo de entrada y salida, respectivamente.
2. **Tipo rep**:

```
class Polinomio {
      private:
            float * p;
            int maxgrado;
            int reservados;
```

3. **Función de abstracción**

$$f_A(r) = r.p[r.maxgrado]x^{r.maxgrado} + r.p[r.maxgrado-1]x^{r.maxgrado-1} + \cdots + r.p[0]$$

4. **Invariante de la representación**:
   a) maxgrado ≥ 0
   b) p tiene reservada, al menos, memoria para $maxgrado + 1$
   c) $p[maxgrado] \neq 0$
   d) $\forall i > maxgrado \longrightarrow p[i] = 0$

□

Ejemplo 3.1.11
T.D.A. **DNI**

1. **Especificación**: secuencia de 8 dígitos seguidos por una letra tal que el dígito más significativo es distinto de cero: $d_7 d_6 \cdots d_0 l$
   a) **Operaciones**:
   - Constructor por parámetros
   - Consulta —Get DNI—

- Modificación —Set—
- Operaciones E/S

2. **Tipo rep**:Podríamos optar por diferentes representaciones. Algunos ejemplos serian los siguientes:

```
class DNI {
      private:
            char * d;

class DNI {
      private:
            int num;
            char letra;

class DNI {
      private:
            char dni[9];
```

Si escogemos esta última representación podemos definir la función de abstracción e invariante de la representación de la siguiente manera.

3. **Función de abstracción**:

$$f_A(D) = D.dni[7]D.dni[6]\cdots D.dni[0]D.dni[8]$$

De forma que $D.dni[8]$ tenemos la letra del dni. Y desde la posicion 0 a la 7 los dígitos.

4. **Invariante de la representación**:
   a) $D.dni[i] \in ['0' - '9'] \forall i = 0 \cdots 6 \bigwedge D.dni[7] \in ['1' - '9']$
   b) $D.dni[8] \in ['A' - 'Z']$

□

Ejemplo 3.1.12
T.D.A. **Secuencia de Números primos**

1. **Especificación**: es una secuencia ordenada de enteros $a_0, a_1, \cdots, a_{n-1}$ tal que $\forall i\ a_i$ es divisible sólo entre 1 y él mismo.
   Operaciones:
   - Constructores: constructor por defecto, constructor con los n primeros primos
   - Consulta: acceder al primo de orden i-ésimo, consultar el numero de primos almacenados
   - Escritura y Lectura de un polinomio por un flujo de entrada y salida, respectivamente.
2. **Tipo rep**:

```
class N_Primo {
      private:
            int *p;
            int n;
```

3. **Función de abstracción**:

$$f_A(r) = r.p[0], r.p[1], \cdots, r.p[r.n-1]$$

4. **Invariante de representación**:
   a) $\forall i, j$ tales que $i < j \longrightarrow r.p[i] < r.p[j], 0 \leq i, j < n$ (esta parte expresa que es una secuencia ordenada)
   b) $\forall i$ $r.p[i]$ es divisible sólo por 1 y por él mismo, $0 \leq i < n$ (condición de que cada numero es primo)
   c) $n \geq 0$
   d) p tiene que tener memoria suficiente para almacenar n enteros (condición para poder almacenar la secuencia).

□

Ejemplo 3.1.13
T.D.A **Traductor**

1. **Especificación**: es un conjunto ordenado de pares de palabra origen, palabras destino. Asi para cada palabra en un idioma origen asociamos un conjunto de palabras, en las que se traduce la palabra origen, en el idioma destino. Entradas posibles en un traductor español-inglés sería:
   - hola; hello; hi;
   - adios;bye

   Operaciones:
   - Constructores: constructor por defecto, iniciando al traductor vacio.
   - Consulta: dada una palabra en el idioma origen obtener todas las traducciones de la palabra en el idioma destino; obtener el numero de entradas del traductor; dada una palabra en el idioma destino obtener todas las palabras en el idioma destino
   - Escritura y Lectura de un traductor por un flujo de entrada y salida, respectivamente.
2. **Tipo rep**:

```
1  struct entrada{
2    string p_origen;
3    vector<string> p_destino;
4  };
5  class Traductor  {
6        private:
7         vector<entrada> palabras;
```

3. **Función de abstracción**:

$$\begin{aligned} f_A(r) = \{&(r.palabras[0].p\_origen; r.palabras[0].p\_destino[0], \ldots, \\ &palabras[0].p\_destino[palabras[0].p\_destino.size()-1]), \\ &\vdots \\ &(r.palabras[r.palabras.size()-1].p\_origen; r.palabras[r.palabras.size()-1].p\_destino[0], \ldots, \\ &[r.palabras.size()-1].p\_destino[r.palabras[0].p\_destino.size()-1])\} \end{aligned}$$

4. **Invariante de representación**:

a) $\forall i, j$ tales que $i < j \longrightarrow r.palabras[i].p\_origen < r.palabras[j].p\_origen, 0 \leq i, j < n$ (esta parte expresa que el traductor está ordenado por la palabra origen)
b) $\forall i\ r.palabras[i].p\_destino.size() > 0$ (condición de que cada palabra origen tiene una palabra destino)

□

3.2 Abstracción por generalización

3.2.1 Funciones plantilla

Generalización: Proceso en el que se extraen características comunes a varios objetos y a continuación se define de una forma comprimida todas las posibles características comunes a estos objetos.
Para ver de una forma más clara el concepto de generalización, supongamos la función *Intercambiar*, que intercambia el valor de dos variables.
La función *Intercambiar* para int y para float sería:

```
void Intercambiar (int &a, int &b) {
      int aux = a;
      a = b;
      b = aux;
}
```

```
void Intercambiar (float &a, float &b) {
      int aux = a;
      a = b;
      b = aux;
}
```

Estas dos funciones son idénticas a excepción del tipo de los parámetros de entrada y del tipo de la variable local. Pero la semántica es idéntica. Para solvertar el problema de que esas funciones sólo sirven para floats e int, si queremos una función intercambiar "universal" podemos hacer una plantilla (o template). En C++ seria de la siguiente forma:

```
template <class T> //T es un objeto plantilla generico
void Intercambiar (T &a, T &b)
{
      T aux = a;
      a = b;
      b = aux;
}

int main ()
{
      float f1=5, f2=7;
      //se instancia a float
      Intercambiar (f1,f2);
      string s1="hola", s2="adios";
      //se instancia string
      Intercambiar(s1, s2);
}
```

T es un tipo genérico no definido, que podría instanciarse a cualquier tipo. De forma que cuando se invoca a la función es cuando T se establece como un tipo concreto y por lo tanto Intercambiar actua

sobre ese tipo. Asi en el main se invocan a la función Intercambiar con tipo float y con tipo string. Para resolver estas posibilidades el compilador, en la precompilación, genera dos funciones con los distintos tipos. Por lo tanto hasta la fase de compilación el compilador no sabrá cuanto espacio ocupa los tipos de los parámetros de entrada, y por lo tanto cuanto ocupa el código objeto.

Ejemplo 3.2.1
Función de ordenación como función plantilla

```
1  template <class T>
2
3  void Ordenar_Seleccion (T *v, int n) //cada elemento de v es de tipo generico T
4  {
5        int i, minimo;
6
7        for (i=0; i<n-1; i++)
8        {
9              minimo = i;
10             for (int j=i+1; j<n; j++)
11                   if (v[j] < v[minimo])
12                         Intercambiar(v[j], v[minimo]);
13       }
14 }
15
16 int main ()
17 {
18       int v_int[] = {5,3,7,15,1,2};
19       Ordena_Seleccion (v_int, 6);
20
21       char v_ch[] = {'e','f','d','a','i'};
22       Ordena_Seleccion (v_ch, 5);
23 }
```

Con esta función podemos ordenar enteros, char, string, etc. La única exigencia para que esta función al instanciarse, a un tipo concreto, funcione es que el tipo al que se instancia tenga definido el operador menor ya que en la linea 11 de la función se tiene que poder comparar un elemento con otro.

□

3.2.2 Clases plantilla

Al igual que las funciones las clases podemos hacerlas clases plantillas. Esta posibilidad es muy útil cuando la clase es una clase contenedora de elementos de otro tipo (contiene una colección de elementos de un mismo tipo). De forma que al hacerla plantilla puede contener diferentes conjuntos que se diferencian por el tipo. Un ejemplo que ilustra este concepto es el siguiente en el que se va a reconsiderar el T.D.A Vector Dinámico haciendo que sea una clase plantilla, y asi se pueda instanciar a ser un vector dinámico por ejemplo de enteros, de reales o de objetos definidos por el usuario, etc.

TDA Vector Dinámico

La clase VD no se compila. No se genera VD.o

```
1  template <class T> //En VD.h
2  class VD
3  {
4        private:
5              T* datos;//zona de memoria para almacenar los datos de tipo T
6              int n; //numero de datos almacenados
7              int reservado; //espacio asignado a datos
8              void resize (int nuevotam);
9              void Copiar (const VD<T> &v);
10             void Liberar ();
11       public:
12              /**
13               * @brief Contructor por defecto y con parametro
14               * @param tam: elementos a reservar para el vector dinamico
15               * @note si no se proporciona un valor para tam se tomara como 10
16               */
17             VD (int tam=10);
18             /**
19              * @brief Constructor de copia
20              * @param original: vector dinamico origen
21              */
22             VD (const VD<T> &original);
23             /**
24              * @brief Destructor. Elimina la memoria asociada al vector dinamico
25              */
26             ~VD ();
27             /**
28              * @brief Operador de asignacion
29              * @param v: vector dinamico fuente
30              * @return una referencia al objeto al que apunta this
31              */
32             VD<T> &operator= (const VD<T> &v);
33             /**
34              * @brief Obtiene el numero de elementos almacenados en el vector dinamico
35              */
36             int size() const {return n;}
37             /**
38               * @brief Consulta y modifica el elemento i-esimo
39               * @param i: posicion del elemento
40               *  @return una referencia al elemento i-esimo del vector dinamico
41              */
42
43             T &operator[] (int i) {return datos[i];}//version no constante
44             const T &operator[] (int i) const {return datos[i];} //version constate
45             /**
46              * @brief Inserta un objeto en la posicion pos del vector dinamico
47                @param d: objeto a insertar
48                @param pos: posicion donde insertar.
49                @pre pos debe estar comprendido entre 0 y size()
50                @post aumenta en uno el vector dinamico
51               */
52             void Insertar (const T &d, int pos);
53             /**
54              * @brief Elimina el elemento en la posicion pos
55              *  @param pos: posicion del elemento a borrar.
56              */
57             void Borrar (int pos);
58 };
59 #include "VD.cpp"
```

Ahora VD.h incluye a VD.cpp

```
//principal.cpp
#include "VD.h"
int main ()
{
      VD<int> vint;
      VD<char> vch(100);
      VD<string> vs;
}
```

No podemos compilar VD y principal por separado porque no podemos saber lo que ocupa en memoria T. Se compila principal.cpp y en la fase de preprocesamiento se incluye el contenido de VD.h y de forma transitiva el contenido de VD.cpp. Esto es asi ya que en VD.h incluimos VD.cpp. El compilador al hacer la instanciacion, sutituye cada T que aparece por el tipo instanciado. En el fichero VD.cpp se realiza la implementación de los métodos. Antes de cada método pondremos **template <class T>**. Para indicar el nombre de la función haremos referencia como siempre al nombre de la clase pero de tipo T. Esto quiere decir que, por ejemplo, para especificar la cabecera del método resize escribiremos
template <class T>
void VD<T> :: resize(int nuevotam)

```
//VD.cpp
//No ponemos #include "VD.h", ya hemos incluido VD.cpp en el VD.h
template <class T>
void VD<T> :: resize(int nuevotam)
{
      T *aux = new T [nuevotam];
      int minimo = (n<nuevotam)?n:nuevotam;
      for (int i=0; i<minimo; i++)
            aux[i] = datos[i];

      reservados = nuevotam;
      n = minimo;
      delete [] datos;
      datos = aux;
}

template <class T>
void VD<T> :: Copiar (const VD<T> &v)
{
      reservados = v.reservados;
      n = v.n;
      datos = new T [reservados];
      for (int i=0; i<n; i++)
            datos[i] = v.datos[i];
}
```

```
26
27  template <class T>
28  void VD<T> :: Liberar()
29  {
30        delete [] datos;
31  }
32
33  template <class T> //en otros sitios el class se sustituye por typename
34  VD<T> :: VD (int tam)
35  {
36        reservados = tam;
37        datos = new T [reservados];
38        n=0;
39  }
40  template <class T>
41  VD<T> :: VD(const VD<T> &original)
42  {
43        Copiar(original);
44  }
45
46  template <class T>
47  VD<T> :: ~VD()
48  {
49        Liberar();
50  }
51
52  template <class T>
53  VD<T> & VD<T> :: operator=(const VD<T> & v)
54  {
55        if (this != &v)
56        {
57              Liberar();
58              Copiar(v);
59        }
60
61        return *this;
62  }
63
64  template <class T>
65  void VD<T> :: Insertar (const T &d, int pos)
66  {
67        /*segun estudios estadisticos, la mejor forma de usar un vector dinamico es
68        que n < reservados/2, y cuando se supere esa cifra hacer un resize de
```

```
69        2*reservados*/
70
71        if (n >= (reservados/2))
72              resize(2*reservados);
73
74        for (int i=n; i > pos; i--)
75              datos[i] = datos[i-1];
76
77        datos[pos] = d;
78        n++;
79  }
80
81  template <class T>
82  void VD<T> :: Borrar (int pos)
83  {
84       for (int i=pos; i<n-1; i++)
85          datos[i] = datos[i+1];
86
87      n--;
88
89      if (n < (reservados/4))
90          resize (reservados/2);
91  }
```

3.3 Ejemplos T.D.A. Plantilla

3.3.1 T.D.A. Conjunto de enteros

Especificación

Un objeto de tipo de dato abstracto *conjunto* es una colección ordenada de elementos de tipo entero en la que no existen elementos repetidos

$$\{a_0, a_1, \cdots, a_{n-1}\} \qquad \text{tales que} \qquad \forall i, j \quad i < j \Longrightarrow a_i < a_j$$

Es decir, es ordenada y no tiene elementos repetidos. Al número de elementos del conjunto se le denomina *cardinal*. Si el cardinal del conjunto es 0 se le denomina *conjunto vacío*.

Tipo rep

Para llevar a cabo la representación usaremos el vector dinámico visto en la sección 3.2.2, instanciándolo a entero.

```
1       class Conjunto {
2           private:
3               VD<int> d;
```

Función de abstracción

$$f_A(r) = \{r.d[0], r.d[1], \cdots, r.d[r.d.size() - 1]\}$$

Invariante de representación

- $r.d[i] < r.d[j] \quad \forall i < j$ [1]

Implementación

```
1  //Conjunto.h
2  #include "VD.h"
3
4  struct pareja {
5      int pos;
6      bool esta;
7  };
8
9  class Conjunto {
10     private:
11         VD<int> d;
12
13     public:
14         /*Como no tenemos memoria dinamica, no tenemos que definir
15         contructor por defecto, de copia o destructor. El que tiene memoria
16         dinamica es VD y ya la gestiona.*/
17
18         int size () const {return d.size();} //devuelve el cardinal
19         //Devuelve si esta un elemento y su posicion, en caso de no
20         //estarlo, devuelve el sitio en el que deberia estar
21         pareja Esta (int x);
22         void Insertar (int x);
23         void Borrar (int x);
24 };
```

```
1  //Conjunto.cpp
2  #include "Conjunto.h"
3
4  pareja Conjunto :: Esta (int x) const {
5      //Al estar ordenado podemos hacer una busqueda binaria
6      int n = d.size();
7      int inicio = 0, fin = n;
8
9      while (inicio < fin) {
```

[1] es decir, no hay elementos repetidos y todos los elementos están ordenados

```
10          int mitad = (inicio + fin)/2;
11          if (d[mitad] == x) {
12              pareja p = {mitad, true};
13              return p;
14          }
15
16          else {
17              if (x > d[mitad])
18                  inicio = mitad + 1;
19
20              else
21                  fin = mitad;
22          }
23
24          pareja p = {inicio, false};
25          return p;
26      }
27  }
28
29  void Conjunto :: Insertar (int x) {
30      pareja p = Esta (x);
31
32      if (!p.esta)
33          d.Insertar (p.pos, x);
34  }
35
36  void Conjunto :: Borrar (int x) {
37      pareja p = Esta (x);
38
39      if (p.esta)
40          d.Borrar (d.pos);
41  }
```

3.3.2 T.D.A. Vector Disperso

Especificación

Un objeto de tipo de dato abstracto *vector disperso* es un array 1-d de parejas formadas por un índice o clave un valor asociado de tipo T. Si el índice que se consulta no está en el array se devuelve un valor por defecto d.

$$\{(i_0, v_{i_0}), (i_1, v_{i_1}), \cdots, (i_{n-1}, v_{i_{n-1}}), (*, d)\}$$

Con $(*, d)$ se representa todos aquellas claves que no están presentes en el vector disperso y que tienen asociado el valor d. Las operaciones mas relevantes de este T.D.A son:

- *T GetDefault() const*: devuelve el valor por defecto (*d*)
- *T Get(int i) const*: obtiene el elemento con índice o clave i
- *void Set(int i,const T v)*: modifica un elemento con clave i poniéndole como valor asociado v.
- *int NumNoDefault()const*: devuelve el numero de elementos con valor diferente al por defecto.
- *void DatosPosicion(int i,int clave, T valor) const*: obtiene la pareja (clave,valor) asociado al elemento almacenado en la posicion i del vector disperso.
- *void Insertar(int ind,const T v)*: inserta en el vector disperso la pareja (ind,v)

Tipo Rep

Para llevar a cabo la representación usaremos el vector dinámico visto en la sección 3.2.2, instanciándolo a Elemento siendo este:

```
1  template <class T>
2  struct Elemento{
3     int clave;
4     T valor;
5  }
6  template <class T>
7  class Vdisperso{
8    private:
9       VD<Elemento> datos;
10      T valor_por_defecto;
11      ...
12   }
```

Función de Abstracción

Sea r un objeto de tipo rep

$$f_A(r) = \{(r.datos[0].clave, r.datos[0].valor), \cdots, (r.datos[r.datos.size()-1].clave, r.datos[r.datos.size()-1].valor), (-1, r.valor_por defecto)_{\forall i \notin [0, r.datos.size()-1}\}$$

Invariante de representación

- $r.valor\_por\_defecto \neq r.datos[i].dato \; \forall 0 \leq i < r.datos.size()$
- $r.datos[i].clave \neq r.d[j].clave \quad \forall i < j$ [2]

Implementación

```
1  //VDisperso.h
2  #include "VD.h"
3  template <class T>
4  struct Elemento {
5      int clave;
6      T valor;
```

[2] es decir, no hay claves repetidas

```
};
template <class T>
class VDisperso {
    private:
        VD<Elemento> d;
                T valor_por_defecto;

                bool PosicionClave(int & pos,int clave)const;
    public:
        /**
          @brief Construye el vector disperso con un valor por defecto
          @param vdef: valor por defecto
          @note si no da ningun valor se obtendra el que devuelve el
           constructor por defecto de la clase T
         */
         VDisperso(const T &vdef=T()){
           valor_por_defecto=vdef;
         }
        /**
         * @brief Devuelve el valor por defecto
         */
        T GetDefault()const{ return valor_por_defecto;}

        /**
         * @brief Obtiene el valor asociado a una clave
         * @param clave: un valor clave
           @return valor asociado a la clave
        */
        T Get(int clave)const;

        /**
         * @brief Modifica el valor clave asociado a la clave
           @param clave: un valor clave
           @param v: valor asociado a la clave.
         */
        void Set(int clave,const T &v);
        /**
          @brief Obtiene el numero de valores clave diferentes al de por defecto
         */
        int NumNoDefault()const{ return datos.size();}

         /**
          @brief Obtiene el Elemento almacenado en una posicion
```

```
50            @param i: posicion sobre la que se consulta
51            @param clave: el valor clave de la posicion. ES MODIFICADO
52            @param valor: valor asociado a la clave. ES MODIFICADO
53            */
54            void DatosPosicion(int i,int &clave,T&valor)const;
55
56            /**
57             @brief inserta un nuevo elemento
58             @param clave: valor de la clave del nuevo elemento
59             @param v: valor asociado del nuevo elemento
60             */
61            void Insertar (int clave,const T &v);
62  };
63  #include "VDisperso.cpp"
```

```
1   //VDisperso.cpp
2   template <class T>
3   bool VDisperso<T>::PosicionClave(int &pos,int clave)const{
4     for (int k=0;k<datos.size();k++){
5        if (datos[k].clave==clave){
6           pos=k;
7           return true;
8        }
9     }
10    return false;
11  }
12  template <class T>
13  T VDisperso<T>::Get(int clave)const{
14   int pos;
15
16   if (PosicionClave(pos,clave))
17     return datos[pos].valor;
18   else
19     return valor_por_defecto
20  }
21  template <class T>
22  bool VDisperso<T>::Set(int clave,const T &v){
23    int pos;
24    if (PosicionClave(pos,clave)){
25      if (v!=valor_por_defecto) datos[pos].valor=v;
26      else datos.Borrar(pos);
27    }
28    else{
29     if (v!=valor_por_defecto){
```

```
30        Elemento<T>a={clave,v};
31        datos.Insertar(a,datos.size());
32     }
33    }
34  }
35  template <class T>
36  bool VDisperso<T>::DatosPosicion(int i,int &clave,T&valor)const{
37
38    if (i>=0 && i<datos.size()){
39      clave=datos[i].clave;
40      valor=datos[i].valor;
41
42    }
43    else{
44       clave=i;
45       valor=valor_por_defecto;
46
47    }
48  }
49
50  template <class T>
51  bool VDisperso<T>::Insertar(int clave,const T &v){
52     if (v!=valor_por_defecto){
53      int pos;
54      if (!PosicionClave(pos,clave)){
55        Elemento a={clave,v};
56        datos.Insertar(a,datos.size());
57      }
58     }
59
60  }
```

A continuación se muestra un ejemplo de uso. En este ejemplo se crea un vector disperso con valor asociado de tipo float y con valor por defecto igual a -1.0. Se inserta 5 elementos y a continuación se va listando segun las peticiones del usuario.

```
1  //principal.cpp
2  #include "VDisperso.h"
3  int main(){
4   VDisperso<float>vdis(-1.0);
5   for (int i=0;i<5;i++)
6    vdis.Insertar(i,i*0.5);
7   while (true){
8      int ind;
```

```
9       cout<<"Dime un indice:";
10      cin>>ind;
11      float valor=vdis.Get(ind);
12      if (valor!=vdis.GetDefault()){
13        cout<<"Valor asociado:"<<valor<<endl;
14      }
15      else{
16       cout<<"Indice no valido"<<endl;
17      }
18   }
19  }
```

3.3.3 T.D.A. Matriz Dispersa

Especificación

Un objeto de tipo de dato abstracto *matriz dispersa* es un array 1-d de tripletas formadas por un fila, columna y el valor asociado de tipo T. Solamente se almacena los valores con un valor diferente a un valor por defecto.

$$\{(i_0, j_0, m_{i_0,j_0}), (i_1, j_1, m_{i_1,j_1}), \cdots, (i_{n-1}, j_{n-1}, m_{i_{n-1},j_{n-1}}), (*, d)\}$$

Con $(*, d)$ se representa todos aquellas entradas en la matriz que no están presentes en el matriz disperso y que tienen asociado el valor d. Las operaciones mas relevantes de este T.D.A son:

- *T GetDefault() const*: devuelve el valor por defecto (d)
- *T Get(int i,int j) const*: obtiene el elemento en la fila i y columna j
- *int numFilas() const*: obtiene el numero de filas
- *int numCols() const*: obtiene el numero de columnas
- *void Set(int i,int j,const T &v)*: modifica un elemento en la posicion (i,j) poniéndole como valor asociado v.
- *int NumNoDefault()const*: devuelve el numero de elementos con valor diferente al por defecto.

Tipo Rep

Para llevar a cabo la representación usaremos el vector dinámico visto en la sección 3.2.2, instanciándolo a Elemento siendo este:

```
1   template <class T>
2   struct Elemento{
3      int row,col;
4      T valor;
5   };
6   template <class T>
7   class Matriz_Dispersa{
8     private:
9        VD<Elemento> m;
```

```
10        T valor_por_defecto;
11        int nf,nc;//numero de filas y columnas
12       ...
13    }
```

Función de Abstracción

Sea r un objeto de tipo rep

$$f_A(r) = \{(r.datos[0].row, r.datos[0].col, r.datos[0].valor), \cdots, (r.datos[r.datos.size()-1].row, r.datos[r.datos.size()-1].col, r.datos[r.datos.size()-1].valor), (-1, r.valor_por_defecto)_{\forall i \notin [0, r.datos.size()-1}\}$$

Invariante de representación

- $r.valor\_por\_defecto \neq r.datos[i].dato \; \forall 0 \leq i < r.datos.size()$
- $0 \leq r.datos[i].row < r.nr \wedge 0 \leq r.datos[i].col < r.nc$

Implementación

```
1  //Matriz_Dispersa.cpp
2
3  template <class T>
4  T Matriz_Dispersa<T>::Get(int i,int j)const{
5      bool pasado=false;
6      for (int k=0;k<m.size() && !pasado;k++){
7        if (m[k].row==i && m[k].col==j) return m[k].dato;
8        else
9           if ((m[k].row==i && m[k].col>j) ||m[k].row>i) return valor_defecto;
10
11     }
12     return valor_defecto;
13 }
14 template <class T>
15 void Matriz_Dispersa<T>::Set(int i,int j, const T &nuevo){
16    if (nuevo==valor_defecto){
17       //lo buscamos
18       int k=0; bool salir=false;
19       for (int k=0;k<m.size() && !salir;k++){
20            if (m[k].row==i && m[k].col==j){
21                m.Borrar(k);//se borra
22            }
23            else if ((m[k].row==i && m[k].col>j) || m[k].row>i ){
24                 salir=true; //no esta
25            }
26            else
27                 k++;
28       }
```

```
29      }
30      else{
31        if (nuevo!=valor_defecto){
32         bool mod=false;
33         for (int k=0;k<m.size() && !mod;k++){
34          if (m[k].row==i && m[k].col==j){
35              m[k].dato=nuevo;
36              mod=true;
37          }
38          else if ((m[k].row==i && m[k].col>j) || m[k].row>i ){
39                  Elemento<T> a={i,j,nuevo};
40                  m.Insertar(a,k);
41                  mod=true;
42          }
43
44        }
45        if (!mod){
46          Elemento<T> a={i,j,nuevo};
47          m.Insertar(a,0);
48        }
49
50       }
51     }
52 }
```

A continuación se muestra un ejemplo de uso. En este ejemplo se crea un matriz dispersa 3×3 con valor por defecto a -1. Se modifican los valores de dos posiciones la (0,0) y la (2,2). A continuacion se muestra la matriz y el numero de valores diferentes al valor por defecto. Se modifica la (0,0) al valor por defecto, se muestra de nuevo la matriz y el numero de valores iguales al valor por defecto.

```
1  //principal.cpp
2  #include "matriz_dispersa.h"
3  int main(){
4   Matriz_Dispersa<int> M(3,3,-1);
5   M.Set(2,2,5); M.Set(0,0,10);
6   for (int i=0;i<M.numFilas();i++){
7     cout<<endl;
8     for (int j=0;j<M.numCols();j++)
9       cout<<M(i,j)<<'\t';
10  }
11
12  cout<<"Numero de no Defaults "<<M.NumNoDefault()<<endl;
13
14  M.Set(0,0,M.GetValorDefecto());
```

```
15  for (int i=0;i<M.numFilas();i++){
16    cout<<endl;
17    for (int j=0;j<M.numCols();j++)
18      cout<<M(i,j)<<'\t';
19  }
20
21  cout<<"Numero de no Defaults "<<M.NumNoDefault()<<endl;
22 }
```

Lectura Avanzada 3.3.1. Plantillas plantillas

Cuando se instancia un vector a enteros,*vector<int>* realmente lo que se esta realizando es *vector<int,std::allocator<int> >*. Esto quiere decir que los contenedores, en particular los vectores de la STL, tienen dos parámetros cuando se usan para definir objetos: el tipo del elemento que almacena, y el tipo del controlador de memoria. A estos controladores se le denominan *allocator*. Los allocators son usados por los contendores, para reservar y liberar memoria. Suponed la siguiente clase:

```
1 template <typename T, class Cont=vector<T> >
2 class MiVector{
3   private:
4    Cont datos;
5   public:
6     void Add (const T & d);
7     //....
8 };
```

Ahora podríamos tener las siguientes declaraciones

```
1 MiVector<int> a; //el contenedor es un vector
2 MiVector<int,list<int> > b; // el contenedor es una lista
```

Este esquema para instanciar es muy flexible pero tiene posibilidades de generar inseguridad. Ya que no se exige ninguna coordinación entre los tipo de los elementos (T) y el tipo del contenedor. Por ejemplo podríamos tener la siguiente declaración

```
1  MiVector<int,list<string> > raro;//error
```

En este caso el tipo del contenedor es diferente del tipo del elemento que se usa por ejemplo en la función *Add*. Para evitar estos problemas, un template puede tomar como parámetro otro template. De esta forma la *clase Mivector* podría declararse de la siguiente forma:

```
1  template <typename T,template<typename> class Cont>
2  class Mivector;
```

En esta declaración el primer parámetro T, es el nombre del tipo. El segundo parametro *Cont*, es un parametro template template. Es el nombre de una clase platilla que tiene un único tipo. Cabe resaltar que no hemos dado un nombre al tipo del parámetro de *Cont*. Así ahora

```
template <typename T, template <typename> class Cont >
class MiVector{
  private:
   Cont<T> datos;
  public:
    void Add (const T & d);
    //....
};
```

Esta declaración asegura la coordinación entre el tipo del elemento y el tipo del contenedor. Ahora podríamos tener declaraciones del tipo

```
MiVector<int,list> a;
MiVector<string,vector> b;
```

Adicionalmente, podemos emplear un argumento por defecto para el template template

```
template <typename T, template <typename> class Cont = vector>
class MiVector {
    //...
};
//...
MiVector<int> a1; // usa Cont por defecto a vector
MiVector<std::string,List> a2; // Cont como List
```