

# TDA EN C++

## TIPOS PREDEFINIDOS

- SON REALMENTE TDA
- TIENEN ASPECTOS COMUNES QUE FACILITAN SU USO
  - El operador '=' permite asignar valores a variables
  - Si se pasan por valor a una función su valor no se altera y cuando la función termina se liberan los recursos ocupados.
  - El operador '+' suele significar añadir
  - Con `cout << v;` escribimos en la salida estándar el contenido de la variable v
  - etc...

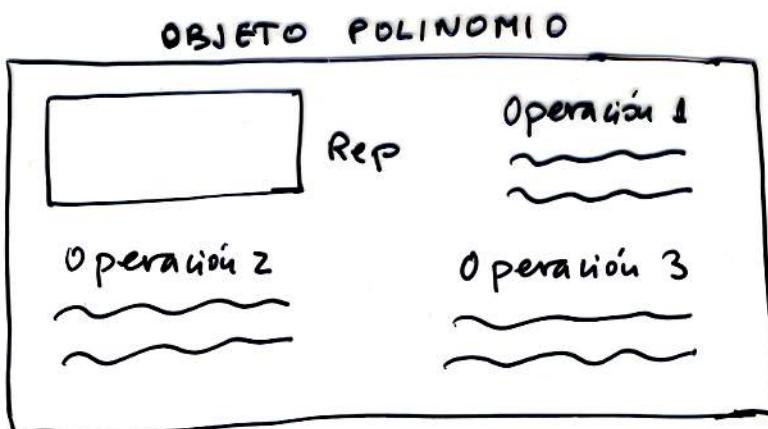
LA IDEA ES INTEGRAR EN EL LENGUAJE LOS NUEVOS TIPOS QUE DEFINIMOS DE LA MISMA FORMA QUE LOS PREDEFINIDOS, PARA QUE CODIGOS COMO ESTE SEAN VALIDOS:

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

int main()
{
    Polinomio p1, p2, res;
    cout << "Introduzca el primer polinomio" << endl;
    cin >> p1;
    cout << "Introduzca el segundo polinomio" << endl;
    cin >> p2;
    res = p1 + p2;
    cout << "La suma de los dos polinomios es " << res << endl;
    return 0;
}
```

## ENCAPSULAMIENTO

TDA = **DATOS** + OPERACIONES



Los objetos encapsulan datos y operaciones

↓  
El nuevo tipo que define el usuario se denominará:

**clase**

y cada una de las instancias de una clase:

**objeto**

## 1. ESTRUCTURAS Y CLASES

Para poder implementar la unión de datos y operaciones se pueden usar las estructuras (struct). A cada uno de sus campos se le denomina miembro.

struct Polinomio {

float \*coef; // Matriz interna con los coeficientes

int grado; // Campo que almacena el grado

int MaxGrado; // Máximo espacio reservado

void AsigCoeficiente (int i, float c); // Asigna al coef. i el valor c

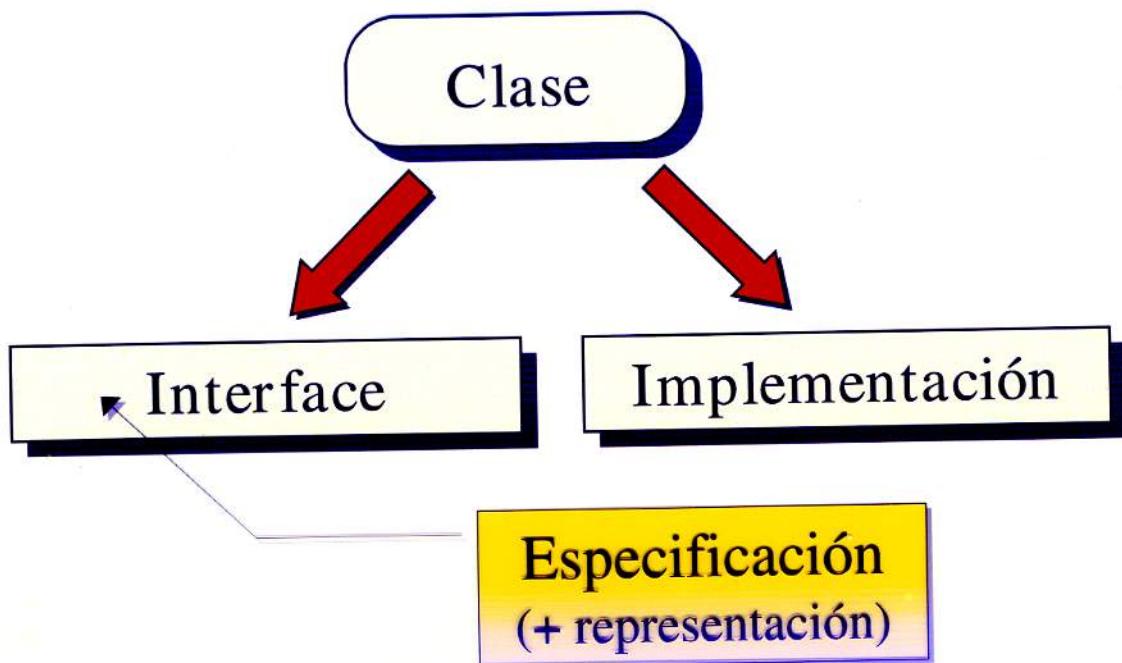
int Grado(); // Devuelve el grado del polinomio

float Coeficiente (int i) // Devuelve el coeficiente de grado i

-----.

}

# Clases en C++



## • Interface

```
class <nombre del tipo> {  
public:  
    // sintaxis de las operaciones de la clase  
    // (cabeceras de las funciones)  
private:  
    // área de datos (representación)  
    // operaciones internas de la clase  
};
```

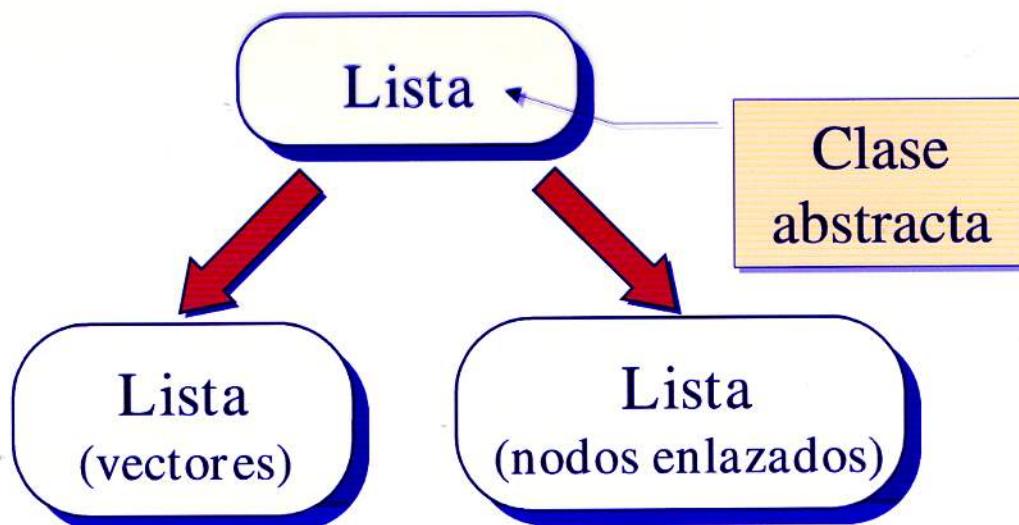
*// sintaxis de las operaciones de la clase  
// (cabeceras de las funciones)*

*// área de datos (representación)  
// operaciones internas de la clase*

¡ Atención !

Por tanto, a priori, en C++ no es posible separar completamente la especificación de la implementación de una clase (la representación se incluye en el interface).

Este problema se puede solventar mediante el uso de la abstracción por especificación. Así, tendremos una clase abstracta que especificará las operaciones del nuevo tipo de dato y, que en general, carecerá de parte de implementación.



## Clase Fecha Versión 0.1

```
class Fecha {  
public:  
    Fecha (int dia, int mes, int anio);  
    void Lee(char * cadena);  
    int Dia() const;  
    int Mes() const;  
    int Anio() const;  
    void Escribe(char * cadena) const;  
    void Siguiente(Fecha * g) const;  
    void Anterior(Fecha * g) const;  
    bool menor(Fecha f2);  
    bool menor_o_igual(Fecha f2);  
  
private:  
    int dia;  
    int mes;  
    int anio;  
};
```

## Control de acceso

Las clases permite controlar el acceso a sus componentes (miembros) usando dos nuevas palabras reservadas:

- **private**: los miembros definidos tras esta indicación sólo son accesibles por miembros de la clase, pero no por nadie externo a la clase. Ocultamiento de información.
- **public**: los miembros definidos son accesibles para cualquiera.

Estas palabras se incluyen en la definición de la clase entre las declaraciones de los miembros. Tienen efecto desde que se declaran hasta que aparece otra o hasta al final. Por defecto, todos los miembros son privados.

- **Implementación**

Consiste de la codificación de los métodos (operaciones) de la clase en función de la representación elegida (recuérdese que en C++ ésta se indica en el interface).

## Ejemplos

```
int racional::numerador () const
{ // retorna el campo numerador de un racional
    return num;
}
int racional::denominador () const
{ // retorna el campo denominador de un racional
    return den;
}
```

Nombre  
cualificado

## Definición de funciones miembro

Las funciones miembro se declaran dentro de la definición de la clase y se definen fuera de ésta.

Para indicar que son miembros de su clase se usa el operador de ámbito (::):

```
int Fecha::Mes() const {  
    return mes;  
}  
  
void Fecha::Siguiente(Fecha & g) {  
    ...  
}
```

La definición de las funciones miembro es parte de la implementación (visión interna del TDA). Por ello, su definición se incluye en el fichero .cpp y no en el .h. El usuario del tipo, no necesita conocer su implementación para usarlas. Su definición sólo es necesaria en tiempo de enlazado, no de compilación.

## Acceso a los miembros

En la definición de una clase se incluyen tanto definiciones de datos (variables de instancia) como operaciones asociadas (funciones miembro o métodos). Esto junto con el control de acceso (ocultamiento de información) favorece el encapsulamiento.

Todas las funciones miembro tienen acceso a todos los miembros de la clase tanto `public` como `private`.

Cada función miembro recibe implícitamente un argumento oculto que referencia al objeto sobre el que se aplica. Se trata de un puntero al objeto de nombre `this`, que no se declara explícitamente pero que siempre está disponible.

El acceso a otros miembros de la clase se hace nombrándolos directamente o a través de `this`:

```
Fecha::Fecha (int D, int M, int A) {  
    dia = D;  
    this->mes = M;  
    anio = A;  
}
```

## Funciones miembro inline

Existen funciones miembro pequeñas que, por cuestiones puramente de *eficiencia* interesa que sean definidas **inline**.

Esto no afecta al usuario del tipo de dato, pero sí al compilador. Es necesario conocer su definición en tiempo de compilación (antes del enlazado). Por ello es conveniente incluir su definición en el fichero cabecera.

Las funciones miembro **inline** se pueden definir:

- a) dentro de la clase
- b) fuera de la clase

## Definición de funciones miembro inline

- a) Dentro de la clase: Su definición se pone justo después de la declaración de la función. No se incluye la palabra reservada `inline`:

```
class Fecha {  
    int Dia() const  
    { return dia; }  
};
```

- b) Fuera de la clase: La definición se hace como en el caso general, pero se precede de la palabra reservada `inline` y se incluye en el fichero ~~cabecera~~, una vez finalizada la definición de la clase:

```
class Fecha {  
    int Dia() const;  
};  
  
inline int Fecha::Dia() const {  
    return dia;  
}
```

## Funciones amigas (friend)

En ocasiones es necesario que una función global (no miembro de una clase) tenga acceso a sus miembros privados. Esto implica saltarse los mecanismos de control de acceso. C++ ofrece una solución para estas situaciones especiales: funciones amigas.

Se tienen que declarar dentro de la definición de la clase, precediendo su definición con la palabra reservada **friend**.

```
class Fecha {  
    friend ostream & operator<<(ostream &s,  
                                const Fecha & f);  
};
```

## Constructores (I)

Son funciones invocadas automáticamente en cuanto se define un objeto de la clase. Su objetivo es poner el objeto en un estado inicial válido.

- No devuelven objetos ni tienen tipo de dato de retorno, ni siquiera `void`.
- Su nombre coincide con el de la clase.
- Puede haber más de un constructor, pero sus declaraciones (prototipos) han de ser distintas.
- Un constructor que no recibe argumentos es el *constructor por defecto*:

```
class racional {  
    racional();  
};
```

Class Rational {

private:

int num;

int den;

[en.h]

public:

Rational();

Rational(int n, int d);

~~~Rational();~~

}

Rational::Rational()

[en.lpp]

{

num = 0;

// this → num = 0

den = 1;

// this → den = 1

}

Rational::Rational(int n, int d)~~;~~

{

assert(d != 0)

num = n;

den = d;

}

Rational r; // crea el rational r=0/1

Rational r(3, 4); // construye el rational r=3/4

Rational::~Rational()

~~=====~~

#### 4. CONSTRUCTORES DE COPIAS

Determina como se copian los parámetros. Si no se define, el compilador asigna una por defecto mediante la copia de cada uno de los miembros de la clase.

##### Forma de definirlo

Es una función con el mismo nombre que el de la clase (**constructor**) y teniendo como parámetro el objeto a copiar (**copia**). Este parámetro:

- Pasa por referencia (lo que entra que se copie)
- se declara constante (el objeto que se copia no debe ser modificado)

Polinomio:: Polinomio (const Polinomio & orig)

```
{ int i;  
    this->MaxGrado = orig.grado;  
    this->coef = new float [this->MaxGrado + 1];  
    for (i = 0; i <= this->MaxGrado; i++)  
        this->coef[i] = orig.coef[i]; // Copiamos los coeficientes  
    this->grado = orig.grado;  
}
```

Nota: El objeto en el que se copia no estaba creado.  
Esta función es un CONSTRUCTOR !!

Racional:: Rational (const Rational & r)

{

assert (r.den != 0) //en realidad no haria falta

num = r.num;

den = r.den;

}

Rational a (3, 4) //a =  $\frac{3}{4}$

Rational b (a); //b = a =  $\frac{3}{4}$

Rational:: Rational (const Rational & r): num (r.num),  
den (r.den) {}

//constructores de miembros

## Destructores

- Son las operaciones antagónicas de los constructores.
- Su objetivo es liberar todos los recursos asociados al objeto (p.ej: memoria).
- No tienen tipo de retorno.
- Su nombre se construye anteponiendo ~ al nombre de la clase:

```
class Fecha {  
    ~Fecha();  
};
```

- Se invocan automáticamente cuando un objeto deja de existir.

## SOBRECARGA DE OPERADORES

Ejemplos previos:

Sumar(a, Dividir(Producto(b, c), Producto(c, Sumar(e+f))))

↓,

$a + (b * c) / (c * (e + f))$

? Por que no poder hacer esto independientemente del tipo de dato del que sean a, b, c, e, f, si esas operaciones tienen sentido? (complejos, matrices, n= racionales...)

↓,

Sobrecarga de operadores

Se realiza añadiendo a la palabra "operator" el operador correspondiente.

Ej: TDD Rational

```
class Rational {  
    private:  
        int numerador, denominador;  
        void simplifica();  
    public:  
        Rational() {numerador=0; denominador=1};  
        Rational operator+(Rational r);  
}
```

dónde la función podría ser:

Racional Rational::operator+ (Rational r)

    { Rational res;

        res.numerador = numerador \* r.denominador +  
                      denominador \* r.numerador;

        res.denominador = denominador \* r.denominador;

        res.simplifica();

        return res;

}

De esta forma, cuando el compilador encuentra  
 $a+b$  con  $a, b$  Racionales interpreta:

a.operator+(b)      (correcto!!)

es decir,

llama a la función miembro de suma sobre el  
objeto a con el parámetro b.

## OPERADORES SOBRECARGABLES

+ - \* / %  $\wedge$   $\&$  | binarios  
~ ! = < > += -= \*=  
/= %= ^= |= << >> >>=  
<<= == != << >= && || ++  
-- >> , -> [] () new delete

new[] delete[]

## OPERADORES NO SOBRECARGABLES

. . \* :: ?:

## 2. OPERADORES COMO FUNCIONES MIEMBRO O COMO AUXILIARES

Ejemplo: sobrecarga del producto de 2 polinomios:

a) como una función miembro:

```
class Polinomio {  
public:  
    ...  
    Polinomio operator*(const Polinomio & p);  
};
```

con lo que hacer  $p * q$  es equivalente a  $p.operator*(q)$ .

b) como una función auxiliar:

```
class Polinomio {  
public:  
    ...  
    friend Polinomio operator*(const Polinomio & p1,  
                               const Polinomio & p2);  
};
```

(función amiga si queremos acceder a la parte privada de la clase)

y definir la función en algún lugar como:

```
Polinomio operator*(const Polinomio & p1, const Polinomio & p2)  
{  
    ...  
}
```

(al ser función auxiliar no se pone  $\text{Polinomio}::\text{Polinomio}$  operator).

¿Son equivalentes? → NO!!!

• Considerar una función como miembro de la clase, obliga a que sea llamada con un objeto de dicha clase:

$$p.\text{operator}*(q) \Rightarrow p * q \quad \text{con } p, q \text{ polinomios}$$

obliga a que  $p$  sea una instancia de la clase Polinomio. Y a veces es imposible para un miembro de la clase tener la operación que queremos: P. ej. multiplicar un float por un polinomio.

Si ponemos  $f * p$  con  $f$  float y  $P$  Polinomio no puede interpretarse como  $f.\text{operator}*(p)$  porque estaríamos considerando un miembro de la clase float que sobrelanza el producto.

### Solución:

Una función auxiliar que al no ser miembro no necesita que el primer parámetro sea de la clase Polinomio:

`Polinomio operator*(float f, const Polinomio& p)`

que se anadiría a la clase indicando que es una función amiga (friend) caso de que necesitáramos alrededor a la parte privada.

## 1. OPERADOR DE ASIGNACION

En C++ el operador de asignacion tiene un valor predefinido: llama a la asignacion de cada uno de los miembros de la clase, pero es posible que lo queramos definir nosotros.

No confundirlo con el constructor de copias



La asignacion da un valor a un objeto que ya estaba construido, mientras que el constructor de copias da un valor a un objeto que està por construir.

La cabecera de este operador es en general:

$\text{X} \& \text{ operator}=(\text{const } \text{X} \& \text{ orig});$

donde X una clase cualquiera

- Recibe un parámetro del tipo de la clase, que se pasa por referencia y no puede ser modificado.
- Devuelve una referencia a un objeto de la clase, es decir, la referencia que devuelve es la del objeto que recibe la petición de asignación: \*this.

↓  
Esto permite usar la asignación dentro de expresiones, ya que el resultado es el objeto de la asignación  $\Rightarrow$  encadenar asignaciones, comparar resultados de asignaciones, etc.

### Ejemplo:

Rational & Rational::operator=(const Rational & orig)

```
{ if ( !&orig != this ) // comprueba si el objeto que se
    this->num = orig.num; // asigna su igual al asignado
    this->den = orig.den;
}
return *this;
```

y

- Como puede verse, el objeto al que se asigna el valor orig ya está creado por lo que ~~se~~ diferencia del constructor de copias.
- Se devuelve una referencia al objeto al que se asigna orig para que pueda ser usado dentro de otra expresión:

$$a.\text{operator}=(b)=c \implies (a=b)=c$$

$a = b = c$

$a.\text{operator} = (b.\text{operator} = (( )) )$

$\downarrow \#$

$a.\text{operator} = (b).\text{operator} = (( ))$

$(a = b) = c$

(porque  $b = ()$ ) !!!

— • —

$x = \text{int}();$

$\downarrow$       llamada al constructor  
 $x = 0;$       de los enteros

— • —

## Operadores de entrada/salida

La característica de sobrecarga de C++ permite proveer a los nuevos tipos de datos de operaciones de entrada/salida de una forma muy simple.

Sólo es necesario sobrecargar los operadores `<<` (para la salida) y `>>` (para la entrada). Los operadores son binarios y requieren como primer argumento un *stream* (excepto si se utilizan como operadores de desplazamiento de bits).

Los objetos de tipo *stream* permiten copiar desde un fichero (entrada) o hacia un fichero (salida).

## *Streams de salida*

El tipo de dato para *streams* de salida se denomina *ostream* (por *output stream*).

```
cout << "La media de " << a << " y " << b << " es "  
<< (a+b)/2 << endl;
```

Dispositivo de salida estándar  
**cout (por *common output*)**

## Formato de salida para los racionales

```
ostream & operator << (ostream & salida,  
                         const racional & r)  
{  
    // imprime un racional en un stream de salida  
    salida << r.numerador() << '/' << r.denominador();  
    return salida;  
}
```

## *Streams* de entrada

El tipo de dato para *streams* de entrada se denomina *istream* (por *input stream*).

cin >> n;

Dispositivo de entrada estándar  
cin (por *common input*)

Cuando se utiliza el operador `>>` en situaciones en que se requiere un valor booleano, el resultado generado se utiliza para indicar el fin de fichero.

```
while (cin >> entero) {  
    // procesar entero  
    .....  
}
```

## Formato de entrada para los racionales

```
istream & operator >> (istream & entrada, racional & r)
{
    // lee un número racional desde el stream de entrada
    int n, d = 1;
    char ch;
    // lee el numerador y el denominador si existe
    entrada >> n
    entrada >> ch;
    if (ch == '/')
        entrada >> d;
    else
        entrada.putback(ch); ◀
    r = racional(n, d);
    return entrada
}
```

La operación `putback` devuelve el último carácter leído al *stream* de entrada.

### 3. OPERADORES DE E/S

Un caso particular de sobrecarga de operadores es el caso de los operadores `>>` y `<<` aplicados sobre los tipos "istream" y "ostream" respectivamente.

`cout << a` es equivalente a `cout.operator<<(a)` que da como resultado el mismo ostream (`cout`) para encadenar, si queremos, más veces el operador con otros parámetros. Al igual que con los demás operadores, podemos sobrecargar `>>` y `<<` con nuestros tipos.

Ejemplo:

Supongamos que queremos añadir operaciones de E/S al tipo Fecha.

El operador se aplicará sobre el tipo istream u ostream es decir, tenemos que sobrelugar los operadores definiendo una función auxiliar, porque no podemos añadir funciones miembro a estas clases.

Además, si no van de acuerdo a la parte privada de la clase no necesitan ser declaradas como funciones amigas.

## Ejemplo:

ostream & operator << (ostream & s, const Fecha & f)

```
{  
    s << f.DiaFecha() << '/' << f.MesFecha() << '/' << f.AñoFecha();  
    return s;  
}
```

istream & operator >> (istream & s, Fecha & f)

```
{  
    int d, m, a;  
    char c;  
  
    s >> d >> c >> m >> c >> a;  
  
    Fecha g(d, m, a); // para no tener que ponerla friend  
    f = g; // s >> f.dia >> f.mes >> f.año  
    // ....  
    // En realidad habría que comprobar  
    // que es una fecha correcta o bien en  
    // tarea la hará el constructor.  
    return s;  
}
```

y usarlos como:

Fecha f;

cout << "Introduzca una fecha" << endl;

cin >> f;

cout << "La fecha " << f << " corresponde a la semana"  
<< f.NumeroSemanaFecha() << endl;

=====

## Parametrización o Generalización

**Parametrización** de un tipo de dato consiste en introducir un parámetro en la definición del tipo para poder usarlo con distintos tipos.

Ejemplo: `VectorDinamico<T>`, donde T puede ser `int`, `complejo`, `polinomio`, etc.

Las especificaciones de `VectorDinamico` de `int`, `float`, `Fecha`, etc. son todas iguales salvo por la naturaleza específica del tipo de elementos a incluir en el vector. En este caso, todo es común (especificación, representación e implementación).

En lugar de escribir cada especificación, representación e implementación independientemente se puede escribir una sola de cada, incluyendo uno o varios parámetros que representan *tipos de datos*. (Es el mismo mecanismo de abstracción que hizo surgir el concepto de procedimiento).

Así, en la especificación, representación e implementación de `VectorDinamico` aparecerá T en lugar de nombres de tipos concretos.

## Parametrización de tipos en C++

El mecanismo que ofrece C++ para parametrizar tipos son los *template* de clases.

Declaración de un template:

`template < parámetros > declaración`

Los parámetros de la declaración genérica pueden ser:

- `class identificador`. Se instancia por un tipo de dato.
- `tipo-de-dato identificador`. Se instancia por una constante.

## Clases template

```
template<class T, int n>
class array_n {
private:
    T items[n];
};

array_n<complejo,1000> w;
```

La definición de las funciones miembros que se hagan fuera de la clase se escriben así:

```
template <class T>
T VectorDinamico<T>::componente(int i) const
```

```
{
    return datos[i];
}
```

template <class T>  
class VectorDinamico{

- - - : private:  
- - - : T \* datos;  
}; : : : ;

## Tratamiento de los templates

Para usar un tipo genérico hay que instanciarlo, indicando los tipos concretos con que se quiere particularizar. Ejemplos:

```
VectorDinamico<int> vi;  
VectorDinamico<float> vf;
```

El compilador generará las definiciones de clases y sus funciones miembro correspondientes para cada instanciación que encuentre. Por ello, la definición completa de la clase genérica debe estar disponible: tanto la definición de la clase como las de las funciones, para que el compilador pueda particularizarlas.

Por ello, el código se organizará como siempre: interfaz en el fichero .h e implementación en el fichero .cpp. Pero la inclusión será al revés. No se incluirá el .h en el .cpp, sino el .cpp al final del .h.

## template: organización del código

VD.h

```
#ifndef __VD_H__  
#define __VD_H__  
  
template <class T>  
class VectorDinamico {  
    ...  
};  
  
#include "VD.cpp"  
#endif
```

VD.cpp

```
#include <cassert>  
  
template <class T>  
VectorDinamico<T>::VectorDinamico (int n)  
{  
    ...  
}
```

## TDA COMPLEJO

```
/**  
 * @file complejo.h  
 * @brief fichero cabecera del TDA complejo  
 *  
 */  
  
#ifndef -complejo-h  
#define -complejo-h  
  
/*  
 * @brief TDA complejo  
 * Una instancia de tipo de dato abstracto dc complejo  
 * es un objeto del conjunto de los numeros complejos,  
 * compuestos por dos valores flotantes que representan,  
 * respectivamente, la parte real e imaginaria.  
 * Lo representamos como:  
 * (r, i)  
 *  
 * Un ejemplo de su uso puede verse en:  
 * @include ejemplo_complejo.cpp  
 *  
 * @author  
 * @date  
 */
```

class complejo {

private:

/\*

- \* `el_pase` representante Rep del TDA Conjunto
- \*
- \* `el_sección` invarianto Invariante de Representación
- \*
- \* El invariante es `le` verdadero
- \*
- \* `el_sección_facil` función de Abstracción
- \*
- \* Un objeto válido `de` rep del TDA Complejo
- \* representa al valor:
- \* (`rep.real`, `rep.imaginaria`)
- \*

\*/

`double v_real; /* < parte real */`

`double v_imaginario; /* < parte imaginaria */`

public:

/\*

- \* `el_brief` constructor de la clase
- \* `el_param` parte real del complejo a construir
- \* `el_param` parte imaginaria del complejo a construir

\*/

`Complejo (double r=0, double i=0): v_real(r), v_imaginario(i) {}`

`// Complejo (const Complejo & c)`

`// ~Complejo ()`

`// operator =`

1/2

\* Objetivo Parte imaginaria

\*  $\text{J}(\text{return})$  Devuelve la parte imaginaria del complejo

\*/

double imaguaria( ) const { return v\_imaginario; }

144

\* **elbrief** Parte real

\* **obtener** parte real  
\* **obtener** devuelve la parte real del complejo

1

double real( ) const { return v\_real; }

}; //fin de définition de class

/ 44

**obrief** Realiza la suma de complejos

param  $c$  es el primer sumando

\* **param** C2 es el segundo sumando

\* return Devuelve la suma de c1+c2

1

inline Complejo operator + (Complejo c1, Complejo c2)

{

```
return complejo (c1.real() + c2.real(),
```

(1.imaginaria() + (2.imaginaria()));

1

/\*\

**ejercicio** Realiza la diferencia de complejos

\*param d es el primer complejo

param C2 es el segundo complejo

\* return Devuelve la diferencia  $c_1 - c_2$

1

inline complejo operator - (complejo c1, complejo c2)

1

return Complex(c1.real() - c2.real(),

(1. *imaginaria* () - 2. *imaginaria* ()) );

/\*  
 \* brief Salida de un complejo a ostream  
 \* param os es el stream de salida  
 \* param c es el complejo a escribir  
 \* post se obtiene en la os la cadena (r, i) con r, i  
 \* los valores de la parte real e imaginaria de la c  
\*/

inline std::ostream & operator << (std::ostream & os,  
 const Complejo & c)

```
{
    return os << '(' << c.real() << ',' << c.imaginaria()
        << ')';
}
```

/\*  
 \* brief Entrada de un complejo desde istream  
 \* param is es el stream de entrada  
 \* param c es el complejo leido  
 \* pre La entrada tiene el formato (r, i) con r, i  
 \* los valores de la parte real e imaginaria  
\*/

inline std::istream & operator >> (std::istream & is, Complejo & c)

```
{
    char caracter;
    double r, i;
    is >> caracter >> r >> caracter >> i >> caracter;
    c = Complejo(r, i);
    return is;
}
```

#endif /\* \_complejo\_h \*/

## Ejemplo: Clase Complejo.

| complejo.h                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             | complejo.h                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>#ifndef _complejo_h #define _complejo_h  #include &lt;iostream&gt;  class Complejo { private:     double v_real;     double v_imaginario; public:     Complejo(double r=0, double i=0): v_real(r),v_imaginario(i)     {         // Complejo (const Complejo&amp; c)         // ~Complejo();         // operator=         double real() const         { return v_real; }         double imaginaria() const         { return v_imaginario; }     };     inline Complejo operator+ (Complejo c1, Complejo c2)     {         return             Complejo(c1.real()+c2.real(),c1.imaginaria()+c2.imaginaria());     } };  #endif</pre> | <pre>complejo.h  inline Complejo operator- (Complejo c1, Complejo c2) {     return         Complejo(c1.real()-c2.real(),c1.imaginaria()-c2.imaginaria()); }  inline std::ostream&amp; operator&lt;&lt; (std::ostream&amp; os, const Complejo&amp; c) {     os &lt;&lt; '(' &lt;&lt; c.real() &lt;&lt; ',' &lt;&lt; c.imaginaria() &lt;&lt; ')'; }  inline std::istream&amp; operator&gt;&gt; (std::istream&amp; is, Complejo&amp; c) {     char caracter;     double r,i;     is &gt;&gt; caracter &gt;&gt; r &gt;&gt; caracter &gt;&gt; i &gt;&gt; caracter;     c = Complejo(r,i);     return is; }  #endif /* complejo.h */</pre> |

```
#include <iostream>           /* Programa de prueba */
#include <complejo.h>
using namespace std;

int main ()
{
    Complejo c1, c2;
    cout << "Introduce un complejo usando el formato
            (r, i) :" << endl;
    cin >> c1;
    cout << "Introduce un complejo usando el formato
            (r, i) :" << endl;
    cin >> c2;
    cout << "La suma es" << c1 + c2 << " y la resta"
        << c1 - c2 << endl;
    cout << "Si sumo 1 al primer complejo da como
    resultado" << c1 + 1 << endl;
    cout << "Si a 1 le resto el segundo complejo da
    como resultado" << 1 - c2 << endl;
    return 0;
}
```