

Abstracción de datos

Joaquín Fernández-Valdivia

Javier Abad

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Granada



Objetivos

- Clarificar el concepto de abstracción y, en particular, el concepto de abstracción de datos
- Aprender a definir abstracciones de datos
 - Especificación
 - Elección de la estructura de datos e implementación de las operaciones
- Aprender a reutilizar el software mediante diferentes mecanismos
- Ver cómo incorpora el lenguaje C++ estos conceptos

Contenidos

- Abstracción en programación
 - Mecanismos de abstracción: parametrización y especificación
- Tipos de abstracción en programación
 - Abstracción procedimental. Especificación de una AP
 - Abstracción de datos. TDA. Visiones de un TDA. Especificación e Implementación de un TDA. Función de Abstracción e Invariante de Representación
 - Abstracción por generalización. TDAs genéricos. Parametrización de tipos en C++
 - Abstracción en iteración

Abstracción en programación

- **Abstracción:** operación intelectual por la que se separa un rasgo o cualidad para analizarlo aisladamente y mejorar su comprensión, ignorando otros detalles.
- **Abstracción en la resolución de problemas:** ignorar detalles específicos buscando el esquema general del problema o de su solución, para obtener una perspectiva distinta.
- **Abstracción:** descomposición en la que se varía el nivel de detalle. Para que sea útil:
 - Todas las partes deben estar al mismo nivel
 - Cada parte debe poder abordarse por separado
 - Las soluciones de las partes deben poder unirse para obtener la solución final del problema

Abstracción en programación

- Nos importa más qué hace un programa, función, o módulo, frente a cómo lo hace.
- El programa, función, o módulo es una caja negra en la que se introduce información (entrada) y se produce nueva información (salida).
- El procesamiento que ocurre entre la información de entrada y de salida no es visible

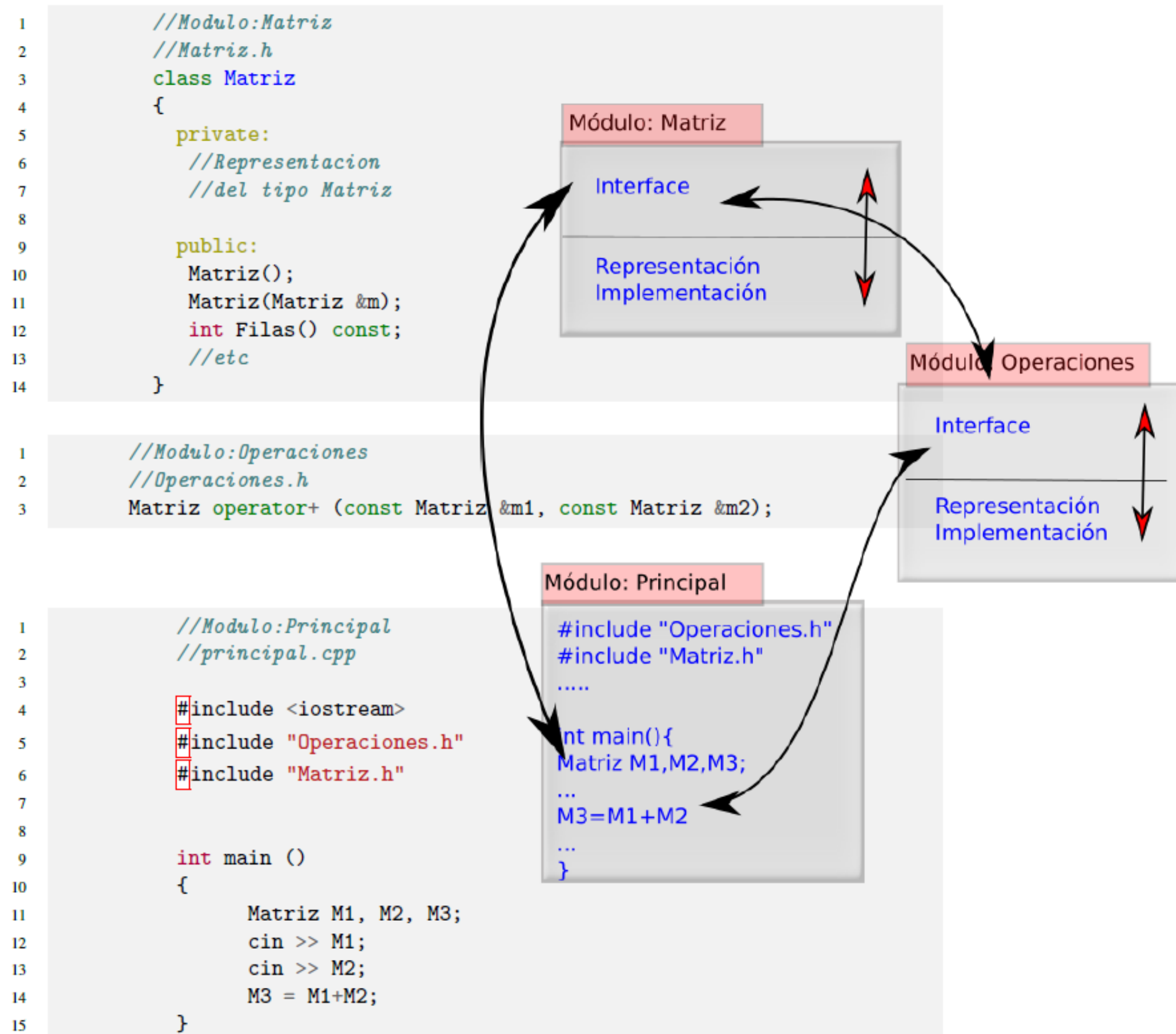
Abstracción en programación

- El usuario de un TDA debe conocer qué operaciones puede realizar sobre él, pero no necesita saber nada acerca de:
 - Forma en como se almacenan los datos
 - Cómo se implementan las operaciones
- El creador del TDA ofrece una parte pública y también una privada.
- Esta parte privada es la que caracteriza el proceso de *ocultamiento de información*.

Ocultamiento de información

- 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.

Ocultamiento de información



Ocultamiento de información

- **Matriz.h** → cabecera de los métodos de la clase Matriz junto con la representación de matriz (datos) (Privada).
- **Matriz.cpp** → implementación de los métodos de la clase Matriz (Privado).
- **Operaciones.h** → cabecera del operador +
- **Operaciones.cpp** → llamar a los métodos de la clase Matriz. Para ello, se comunicará con la interfaz de Matriz para solicitar lo que necesite (Privado).
- **Principal.cpp** → necesita de Matriz y Operaciones, para ello usará la interfaz que ambos módulos le ofrecen.

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.

Mecanismos de abstracción en programación

- **Abstracción por parametrización:** se introducen parámetros para abstraer un número infinito de computaciones.

Ej: `sqrt(valor)`

- **Abstracción por especificación:** permite abstraerse de la implementación de un subprograma asociándole una descripción precisa de su comportamiento

Ej: `double sqrt(double a);`

Requisitos: $a \geq 0$

Efecto: devuelve una aproximación de \sqrt{a}

La especificación es un comentario lo suficientemente definido y explícito como para poder usar el subprograma sin tener que conocer otros elementos

Abstracción por especificación

Suele expresarse en términos de:

- **Precondiciones:** condiciones necesarias y suficientes para que el subprograma se comporte como se ha previsto
- **Postcondiciones:** enunciados que se suponen ciertos tras la ejecución del subprograma si se cumplieron las precondiciones

```
int busca_minimo(int *vector, int tam)
/*
  precondición:
    tam > 0
    vector es un vector con tam componentes
  postcondición:
    devuelve la posición del mínimo del vector
*/
```

Abstracción por especificación

- Debemos dar información sobre:
 - Características sintácticas: cabecera de la función compuesta por (1) **Nombre** de la función, (2) **Parámetros de entrada** y sus tipos, (3) **Tipos de los resultados**.
 - Características semánticas: qué hace y qué devuelve la función con lenguaje natural. **Qué hace sí importa, cómo lo hace no.**

Ejemplo: operador + de dos matrices.

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

Tipos de abstracción en programación

- **Abstracción procedimental o funcional:** definimos un conjunto de operaciones (procedimiento, subprograma, método) que se comporta como una única operación simple.
- **Abstracción de datos (TDA):** definimos un conjunto de datos y una serie de operaciones que caracterizan el comportamiento del conjunto. Las operaciones están vinculadas a los datos del tipo.
- **Abstracción por generalización y abstracción de iteración:** nos permiten trabajar sobre colecciones de objetos sin preocuparnos por el tipo de dato base que contienen, ni por la forma concreta en la que se organizan.

Abstracción procedimental

Permite abstraer un conjunto de operaciones de cómputo como una operación simple. Realiza la aplicación de un conjunto de entradas en las salidas, con posible modificación de las entradas.

- La identidad de los datos no es relevante para el diseño. Sólo nos interesa su número y tipo
- Con la abstracción por especificación, la implementación resulta irrelevante. Lo que importa es qué hace, no cómo lo hace.
 - **Localidad:** Para implementar una abstracción procedimental no es necesario conocer la implementación de otras abstracciones de las que se haga uso
 - **Modificabilidad:** Podemos cambiar la implementación de la abstracción procedimental sin que afecte a otras que la usen, siempre y cuando no modifiquemos la especificación

Especificación de una abstracción procedimental

- **Cabecera (parte sintáctica):** indica el nombre del subprograma y el número, orden y tipo de las entradas y salidas.

Se suele adoptar la sintaxis de un lenguaje de programación. Ej: los prototipos de funciones en C++

- **Cuerpo (parte semántica):** compuesto por

1. Argumentos (parámetros)

2. Requiere

Precondiciones

3. Valores de retorno

4. Efecto

5. Excepciones

Postcondiciones

Especificación de una abstracción procedimental

1. **Argumentos:** explica el significado de cada parámetro de la abstracción procedimental (no el tipo, ya especificado en la cabecera)
 - Indicar restricciones sobre el conjunto de datos sobre los que puede operar el procedimiento
 - Indicar si cada argumento es modificado o no. Cuando un parámetro se modifique se indicará con “ES MODIFICADO”
2. **Requiere:** restricciones derivadas del uso del procedimiento no recogidas en Argumentos. Pej.: la necesidad de que previamente se haya ejecutado otra abstracción procedimental

Especificación de una abstracción procedimental

3. **Valores de retorno:** descripción de qué valores devuelve la abstracción procedimental y qué significan
4. **Efecto:** Describe el comportamiento para las entradas válidas (según los requisitos). Deben indicarse las salidas generadas y las modificaciones producidas sobre los argumentos marcados como “ES MODIFICADO”
 - No se indica nada sobre el comportamiento del procedimiento cuando la entrada no cumple los requisitos
 - No se indica nada sobre cómo se realiza el efecto
5. **Excepciones:** Describe (si es necesario) el comportamiento cuando se da una circunstancia que no permita su finalización con éxito

Ejemplo

```
int busca_minimo(int *vector, int tam)
```

```
/*
```

Argumentos:

vector: array 1-D en el que hacer la búsqueda

tam: número de elementos de vector

Devuelve:

índice del mínimo en el vector

Precondiciones:

tam > 0

vector es un vector con tam componentes

Efecto:

Busca el elemento más pequeño en el array vector
y devuelve su posición, es decir, el valor

$i / v[i] \leq v[j]$ para $0 \leq j < tam$

```
*/
```

Especificación junto al código fuente

- Falta de herramientas para soportar y mantener el uso de especificaciones ➡ responsabilidad exclusiva del programador
- Necesidad de vincular código y especificación
- Incluirla entre comentarios en la parte de interfaz del código
- Usar una herramienta:
 - doc++ (<http://docpp.sourceforge.net>) Permite generar documentación de forma automática en distintos formatos (html, LaTeX,...) a partir del código fuente
 - doxygen (<http://www.doxygen.org>)

Especificación usando doxygen

- Toda la especificación se encierra entre `/** ... */`

```
/**  
    ... texto ...  
*/
```

- Se incluye una frase que describa toda la función: `@brief`
- Cada argumento va precedido de `@param`
- Los requisitos adicionales van precedidos de `@pre`
- Los valores de retorno van precedidos de `@return`
- La descripción del efecto sigue a `@post`

Especificación usando doxygen

```
/**  
@brief Calcula el índice del elemento máximo de un vector  
@param vector array 1-D en el que hacer la búsqueda  
@param tam número de elementos del array \a vector  
  
@pre \a vector es un array 1-D con al menos \a tam componentes  
@pre \a tam > 0  
@return el elemento más pequeño en el array \a vector, es decir,  
el valor  $i$  tal que  $v[i] \leq v[j]$  para  $0 \leq j < \text{\a tam}$   
*/  
  
int IndiceMaximo(int *vector, int tam);
```

Especificación usando doxygen

Function Documentation

◆ IndiceMaximo()

```
int IndiceMaximo ( int * vector,  
                  int  tam  
                  )
```

Calcula el índice del elemento máximo de un vector.

Parameters

vector array 1-D en el que hacer la búsqueda

tam número de elementos del array *vector*

Precondition

vector es un array 1-D con al menos *tam* componentes

tam > 0

Returns

el elemento más pequeño en el array *vector*, es decir, el valor *i* tal que $v[i] \leq v[j]$ para $0 \leq j < tam$

Abstracción de datos

- **Tipo de Dato Abstracto (TDA):** Entidad abstracta (especificación independiente de la implementación) formada por un conjunto de datos y una serie de operaciones asociadas

Ej: los tipos de datos de C++

- 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.

Abstracción de datos

Conceptos:

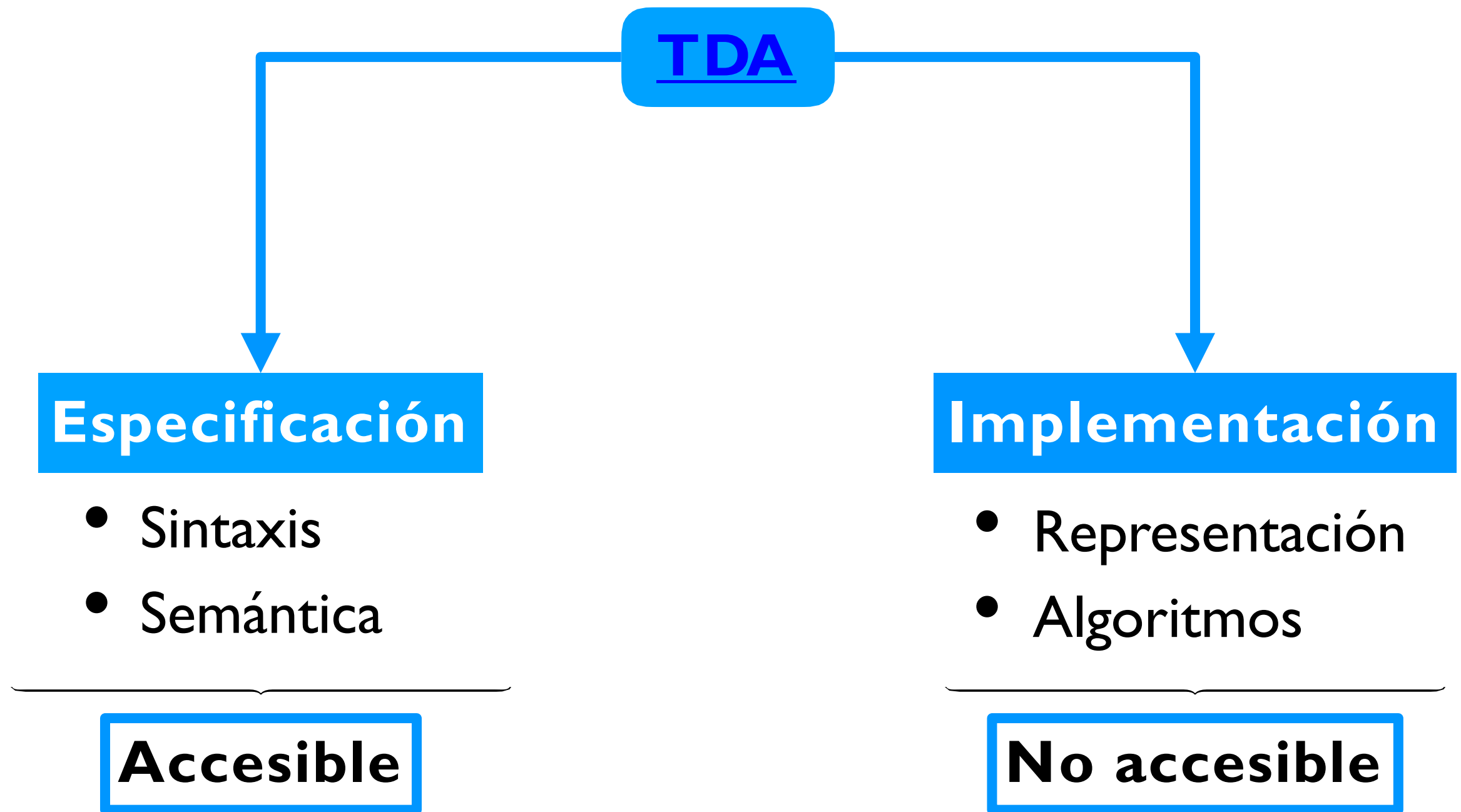
- **Especificación:** descripción del comportamiento del TDA
- **Representación:** forma concreta en que se representan los datos en un lenguaje de programación para poder manipularlos
- **Implementación:** forma específica en que se desarrollan las operaciones

Abstracción de datos

- A tener en cuenta:
 - **Es un tipo de dato:** el hecho de construir un nuevo tipo de dato abstracto nos va a ofrecer una nueva clase de objetos que se traducirá en uno o varios nuevos tipos definidos por el usuario.
 - **Es una abstracción:** trabajamos a nivel de información superior, obviando los detalles que no son relevantes cuando se usa el nuevo tipo.

Visiones de un TDA

- Hay dos visiones de un TDA:
 - Visión externa: especificación
 - Visión interna: representación e implementación
- Ventajas de la distinción de visiones:
 - Se puede cambiar la visión interna sin afectar a la visión externa
 - Facilita la labor del desarrollador, permitiéndole centrarse en cada fase por separado



Especificación de un TDA

- La especificación es **esencial**. Define su comportamiento, pero no dice nada sobre su implementación
- Indica el tipo de entidades que modeliza, qué operaciones se les puede aplicar, cómo se usan y qué hacen
- Estructura de la especificación:

A. Cabecera: nombre del tipo y listado de las operaciones

B. Definición: descripción del comportamiento sin indicar la representación. Se debe indicar si es mutable o no. También se indica dónde residen los objetos

C. Operaciones: especificar las operaciones una a una como abstracciones procedimentales

Especificación de un TDA

- Hay que tener en cuenta:
 - Los objetos que usamos: conlleva definir el dominio en donde tomará valores una entidad que pertenezca a la nueva clase de objetos
 - Cómo usar los objetos: especificando las operaciones una por una usando abstracción funcional
 - Cómo hacer referencia a una operación (*especificación sintáctica*)
 - Qué significado o consecuencia tiene cada operación (*especificación semántica*)

TDA Fecha

/*

TDA Fecha

Fecha: constructor, dia, mes, año, siguiente, anterior, escribe, lee, menor, menor o igual

Definición:

Fecha representa fechas según el calendario occidental

Son objetos mutables

Residen en memoria estática

Operaciones:

*/

TDA Fecha

```
/**
@brief Constructor primitivo.
@param f Objeto creado. Debe ser nulo.
@param dia dia de la fecha. 0<dia<= 31.
@param mes mes de la fecha. 0<mes<= 12.
@param anio año de la fecha.
@pre Los tres argumentos deben representar una fecha válida
según el calendario occidental.
@return Objeto Fecha correspondiente a la fecha dada por los argumentos.
@doc
Crea un objeto Fecha a partir de los argumentos.
Devuelve el objeto creado sobre f.
*/
void constructor (Fecha & f, int dia, int mes, int anio);

/**
@brief Lee una fecha de una cadena.
@param f Objeto creado. Es MODIFICADO.
@param cadena Cadena de caracteres que representa
una fecha en formato dd/mm/aaaa.
@return Objeto Fecha que representa la fecha leída en cadena.
*/
void lee(Fecha & f, char cadena[]);
```


TDA Fecha

```
/**  
@brief Obtiene el día del objeto receptor.  
@param f Objeto receptor.  
@return Día del objeto f.  
*/  
int dia(Fecha f);
```

```
/**  
@brief Obtiene el mes del objeto receptor.  
@param f Objeto receptor.  
@return Mes del objeto f.  
*/  
int mes(Fecha f);
```

```
/**  
@brief Obtiene el año del objeto receptor.  
@param f Objeto receptor.  
@return Año del objeto f.  
*/  
int anio(Fecha f);
```

TDA Fecha

```
/**
@brief Escribe el objeto receptor en una cadena.
@param f Objeto receptor.
@param cadena Cadena que recibe la expresión de f.
Debe tener suficiente espacio.
Es MODIFICADO.
@return Cadena escrita.
@doc
Sobre 'cadena' se escribe una representación en formato 'dd/mm/aaaa'
del objeto f. Devuelve la dirección de la cadena escrita.
*/
char * Escribe (Fecha f, char * cadena);

/**
Cambia f por la fecha siguiente a la que representa.
@param f Objeto receptor. Es MODIFICADO.
*/
void Siguiente(Fecha f, Fecha &g);

/**
@brief Cambia f por la fecha anterior a la que representa.
@param f Objeto receptor. Es MODIFICADO.
*/
void Anterior(Fecha & f);
```

TDA Fecha

```
/**  
@brief Decide si f1 es anterior a f2.  
@param f1  
@param f2 Fechas que se comparan.  
@return  
true, si f1 es una fecha estrictamente anterior a f2.  
false, en otro caso.  
*/  
bool menor(Fecha f1, Fecha f2);
```

```
/**  
@brief Decide si f1 es anterior o igual que f2.  
@param f1  
@param f2 Fechas que se comparan.  
@return  
true, si f1 es una fecha anterior o igual a f2.  
false, en otro caso.  
*/  
bool menor_o_igual(Fecha f1, Fecha f2);
```

TDA Fecha. Ejemplo de uso

```
#include <iostream>
#include "fecha.h"
using namespace std;
int main() {
    Fecha f, g;
    int dia, mes, anio;
    char c1[100], c2[100];

    std::cout << "Introduzca el día del mes que es hoy: ";
    cin >> dia;
    cout << "Introduzca el mes en el que estamos: ";
    cin >> mes;
    cout << "Introduzca el año en el que estamos: ";
    cin >> anio;

    constructor(f, dia, mes, anio);
    Siguiente(f, g);
    if (menor(f, g)){
        Escribe(f, c1);
        Escribe(g, c2);
        cout << "El día " << c1 << " es anterior a " << c2 << endl;
    }
    return 0;
}
```

Implementación de un TDA

- Implica dos tareas:
 1. Diseñar la representación que se dará a los objetos
 2. Basándose en la representación, implementar cada operación
- Dentro de la implementación habrá dos tipos de datos:
 - **Tipo Abstracto:** definido en la especificación con operaciones y comportamiento definido
 - **Tipo *rep*:** tipo a usar para representar los objetos del tipo abstracto y sobre el que implementar las operaciones

Representación del TDA

- Tipo abstracto: TDA Fecha

- Tipo *rep*:

```
struct Fecha {  
    int dia;  
    int mes;  
    int anio;  
};
```

El tipo *rep* no está definido unívocamente por el tipo abstracto.

Pej.: `typedef int Fecha[3];`

TDA Polinomio $(a_n x^n + \dots + a_1 x + a_0)$

Tipo *rep*: `typedef float polinomio[n+1];`

Representación del TDA

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 día, 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:

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

b) Posibilidad 2:

```
1  class Fecha {  
2      private:  
3          string f;
```

c) Posibilidad 3:

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

Implementación del TDA

En toda implementación hay dos elementos importantes:

- **Función de Abstracción:** conecta el tipo abstracto y el tipo *rep*
- **Invariante de representación:** condiciones que caracterizan los objetos del tipo *rep* que representan objetos abstractos válidos

Siempre existen, aunque habitualmente no se es consciente de su existencia

Función de abstracción

- La FA es una función que transforma el tipo rep escogido con el TDA dado en la especificación.
- El tipo rep puede contener muchos más datos que los que se usan en la especificación del TDA.
- El dominio del tipo rep es un superconjunto del conjunto definido en la especificación del TDA.
- La FA establece qué datos, definidos por el tipo rep, son los usados para expresar el TDA definido en la especificación.

$$f_A : rep \longrightarrow A$$

Función de abstracción

- Ejemplos:

- TDA Racional

$$\{\text{num}, \text{den}\} \rightarrow \frac{\text{num}}{\text{den}}$$

- TDA Fecha

$$\text{dia}, \text{mes}, \text{anio} \rightarrow \text{dia/mes/anio}$$

- TDA Polinomio

$$r[0..n] \rightarrow r[0] + r[1]x + \cdots + r[n]x^n$$

Función de abstracción

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 representación (IR)

- Expresión lógica que indica si un objeto del tipo *rep* es un objeto del tipo abstracto o no

Ejemplos:

- TDA Racional: dado el objeto $rep\ r=\{num,den\}$, debe cumplir $den \neq 0$
- TDA Fecha: dado el objeto $rep\ f=\{dia,mes,anio\}$ debe cumplir
 - $1 \leq dia \leq 31$
 - $1 \leq mes \leq 12$
 - Si mes es 4,6,9 u 11, entonces $dia \leq 30$
 - Si mes es 2 y $bisiesto(anio)$, entonces $dia \leq 29$
 - Si mes es 2 y $\neg bisiesto(anio)$, entonces $dia \leq 28$

Indicando la FA y el IR

- Tanto la Función de Abstracción como el Invariante de Representación **deben aparecer escritos** en la documentación

```
#include "racional.h"  
/*
```

```
  ** Función de abstracción:
```

```
  -----
```

```
      fA : tipo_rep ----> Q  
          {num, den} ----> q
```

La estructura {num, den} representa el racional $q = \text{num}/\text{den}$.

```
  ** Invariante de Representación:
```

```
  -----
```

Cualquier objeto del tipo_rep, {num, den}, debe cumplir:
den!=0

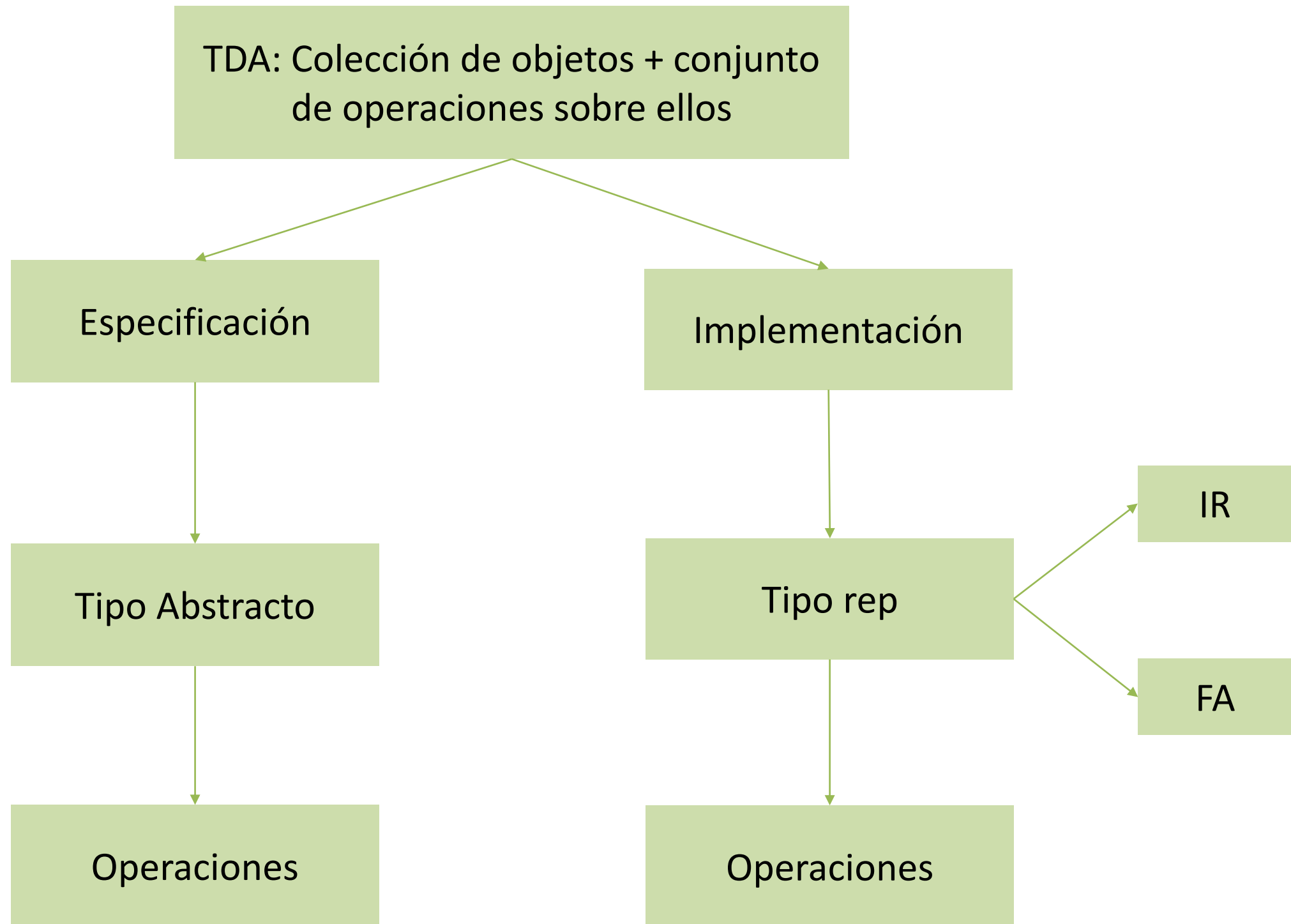
```
*/
```

Preservación del IR

- La conservación del Invariante de Representación es fundamental para todos los objetos modificados por las operaciones que los manipulan. Su conservación se puede establecer demostrando que:
 1. Los objetos creados por los constructores lo verifican
 2. Las operaciones que modifican los objetos los dejan en un estado que verifica el IR antes de finalizar

¡Ojo! Esto sólo se puede garantizar si hay ocultamiento de información

Conceptos asociados a un TDA



TDA Racional

► Tipo abstracto

Pareja de valores (num,den) que representa un número racional num/den, con den ≠ 0

► Tipo *rep*

```
typedef int Racional[2];  
typedef struct{ int numerador;  
               int denominador;  
            } Racional;
```

FA $r \rightarrow \{r.\text{num}, r.\text{den}\} \rightarrow \frac{r.\text{num}}{r.\text{den}}$

IR $r.\text{den} \neq 0$

TDA Fecha

- ▶ Tipo abstracto: Representa fechas según el calendario gregoriano en el formato dd/mm/aaaa

- ▶ Tipo *rep*

```
typedef int Fecha[3];
```

```
typedef struct{ int dia;  
               int mes;  
               int anio;  
            } Fecha;
```

FA $r \rightarrow r.dia/r.mes/r.anio$

IR Un objeto $f=\{dia,mes,anio\}$ debe cumplir

- $1 \leq dia \leq 31$
- $1 \leq mes \leq 12$
- Si mes es 4,6,9 u 11, entonces $dia \leq 30$
- Si mes es 2 y $bisiesto(anio)$, entonces $dia \leq 29$
- Si mes es 2 y $!bisiesto(anio)$, entonces $dia \leq 28$

donde $bisiesto(i) = ((i\%4==0) \ \&\& \ (i\%100!=0)) \ || \ (i\%400==0)$

TDA Polinomio

T.D.A. Polinomio

1. **Especificación:** sucesión de reales a_0, a_1, \dots, a_n que representan polinomios con coeficientes reales del tipo $a_n x^n + a_{n-1} x^{n-1} + \dots + a_0$

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:**

```
1 class Polinomio {  
2     private:  
3         float * p;  
4         int maxgrado;  
5         int reservados;
```

3. **Función de abstracción**

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

4. **Invariante de la representación:**

- a) $maxgrado \geq 0$
- b) p tiene reservada, al menos, memoria para $maxgrado + 1$
- c) $p[maxgrado] \neq 0$
- d) $\forall i > maxgrado \longrightarrow p[i] = 0$

TDA DNI

T.D.A. DNI

1. **Especificación:** secuencia de 8 dígitos seguidos por una letra tal que el dígito más significativo es distinto de cero: $d_7d_6 \cdots d_0l$

a) **Operaciones:**

- Constructor por parámetros
- Consulta —Get DNI—
- Modificación —Set—
- Operaciones E/S

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

```
1  class DNI {
2      private:
3          char * d;
4
5  class DNI {
6      private:
7          int num;
8          char letra;
9
10 class DNI {
11     private:
12         char dni[9];
```

TDA DNI

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 posición 0 a la 7 los dígitos.

4. **Invariante de la representación:**

$$a) D.dni[i] \in ['0' - '9'] \forall i = 0 \cdots 6 \wedge D.dni[7] \in ['1' - '9']$$

$$b) D.dni[8] \in ['A' - 'Z']$$

TDA Números Primos

T.D.A. Secuencia de Números primos

1. **Especificación:** es una secuencia ordenada de enteros a_0, a_1, \dots, 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:**

```
1  class N_Primo {
2      private:
3          int *p;
4          int n;
```

3. **Función de abstracción:**

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

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).

TDA Traductor

T.D.A Traductor

1. **Especificación:** es un conjunto ordenado de pares de palabra origen, palabras destino. Así 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 vacío.
- Consulta: dada una palabra en el idioma origen obtener todas las traducciones de la palabra en el idioma destino; obtener el número 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;
```

TDA Traductor

3. Función de abstracción:

$$f_A(r) = \{(r.palabras[0].p_origen; r.palabras[0].p_destino[0], \dots, \\ palabras[0].p_destino[palabras[0].p_destino.size() - 1]), \\ \vdots \\ (r.palabras[r.palabras.size() - 1].p_origen; r.palabras[r.palabras.size() - 1].p_destino[0], \dots, \\ [r.palabras.size() - 1].p_destino[r.palabras[0].p_destino.size() - 1])\}$$

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)

TDA: clases de operaciones

- **Constructores primitivos:** crean objetos del tipo de dato sin requerir un objeto del mismo tipo como entrada
- **Constructores:** crean objetos del tipo de dato a partir de otro objeto del tipo de dato que reciben como entrada
- **Modificadores:** modifican los objetos del tipo de dato
- **Observadores:** reciben como entrada objetos del tipo de dato y devuelven resultados de otros tipos
- **Destructores:** permiten eliminar un objeto del tipo de dato, recuperando los recursos consumidos para su representación

Abstracción por generalización

- *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.
- La función Intercambiar para int y float sería:

```
1 void Intercambiar (int &a, int &b) {  
2     int aux = a;  
3     a = b;  
4     b = aux;  
5 }
```

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

Abstracción por generalización

- ¿Cómo podríamos conseguir no tener que replicar código simplemente porque cambie el tipo de los datos?
- Funciones plantilla → Templates en C++

```
1  template <class T> //T es un objeto plantilla generico
2  void Intercambiar (T &a, T &b)
3  {
4      T aux = a;
5      a = b;
6      b = aux;
7  }
8
9  int main ()
10 {
11     float f1=5, f2=7;
12     //se instancia a float
13     Intercambiar (f1,f2);
14     string s1="hola", s2="adios";
15     //se instancia string
16     Intercambiar(s1, s2);
17 }
```

Abstracción por generalización

- **T** es un tipo genérico no definido, que podría instanciarse a cualquier tipo.
- Cuando se invoca a la función es cuando **T** se establece como un tipo concreto y por lo tanto Intercambiar actúa sobre ese tipo.
- 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 cuánto espacio ocupa los tipos de los parámetros de entrada, y por lo tanto cuánto ocupa el código objeto.

Abstracción por generalización


- Ordenación por Selección

```
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 }
```

Abstracción por generalización

- Ordenación por Selección

```
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 }
```



Parametrización/generalización de tipos

- La parametrización de un tipo de dato consiste en introducir un parámetro en su definición para poder usarlo con diferentes tipos base

P.ej.: `VectorDinamicoT`, donde `T` puede ser `int`, `complejo`, `polinomio`...

- Las especificaciones de `VectorDinamico` de diferentes tipos base son iguales, salvo por la naturaleza específica de los elementos que contendrá
- En lugar de escribir cada especificación, representación e implementación, se puede escribir una sola, incluyendo parámetros que representan tipos de datos

TDA_s genéricos

- La especificación de un TDA genérico es igual que la de un TDA normal, salvo que se añaden requisitos adicionales para los tipos sobre los que se desee instanciar el TDA, normalmente la existencia de ciertas operaciones.

/**

VectorDinamico::VectorDinamico, ~VectorDinamico, redimensionar, dimension, componente, asignar_componente

Este TDA representa vectores de objetos de la clase T cuyo tamaño puede cambiar en tiempo de ejecución. Son mutables.

Residen en memoria dinámica.

Requisitos para la instanciación:

La clase T debe tener definidas las siguientes operaciones:

- Constructor por defecto
- Constructor de copia
- Operador de asignación

*/

Parametrización de tipos en C++

- C++ ofrece el mecanismo de los *templates* de clases para parametrizar tipos

- 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

Parametrización de tipos en C++

- Definición de funciones miembro:

```
template <class T>
class VectorDinamico{
    private:
        T * datos;
        ...
}
```

```
template <class T>
T VectorDinamico<T>::componente(int i) const{
    return datos[i]
}
```

Parametrización de tipos en C++

- Para usar un tipo genérico hay que instanciarlo, indicando los tipos base concretos que queremos utilizar.

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

- El compilador genera las definiciones de clases y los métodos correspondientes para cada instanciación que encuentre de la clase genérica. Por ello la definición completa de la clase genérica debe estar disponible (definición y métodos)
- Primera solución: el código se organiza como siempre (interfaz en el fichero .h e implementación en el .cpp), **pero la inclusión se hace al revés**. No se incluye el .h en el .cpp, sino el .cpp al final del .h

Parametrización de tipos en C++

- No podemos compilar `VectorDinamico.cpp` y `main.cpp` por separado porque no podemos saber lo que ocupa en memoria `T`.
- Se compila `main.cpp` y en la fase de preprocesamiento se incluye el contenido de `VectorDinamico.h` y de forma transitiva el contenido de `VectorDinamico.cpp`.
- Esto es así ya que en `VectorDinamico.h` incluimos `VectorDinamico.cpp`. El compilador al hacer la instanciación, sustituye cada `T` que aparece por el tipo instanciado.
- En el fichero `VectorDinamico.cpp` se realiza la implementación de los métodos.
- Antes de cada método pondremos `template <class T>`.

Parametrización de tipos en C++

Vector_Dinamico.h

```
#ifndef VectorDinamico_h
#define VectorDinamico_h

template <class T>
class Vector_Dinamico{
...
};

#include "Vector_Dinamico.cpp"
#endif //VectorDinamico_h
```

Vector_Dinamico.cpp

```
#include <cassert>

template <class T>
VectorDinamico<T>::VectorDinamico (int n){
...
}

...
```

Parametrización de tipos en C++

Vector_Dinamico.h

```
#ifndef VectorDinamico_h
#define VectorDinamico_h

template <class T>
class Vector_Dinamico{
...
};

#include "Vector_Dinamico.cpp"
#endif //VectorDinamico_h
```

Vector_Dinamico.cpp

```
#include <cassert>

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



Antes de cada método

Parametrización de tipos en C++

- Segunda solución: separar declaración y definición de la clase en dos ficheros (.h y .hpp), como siempre, y forzar la instanciación con un nuevo fichero

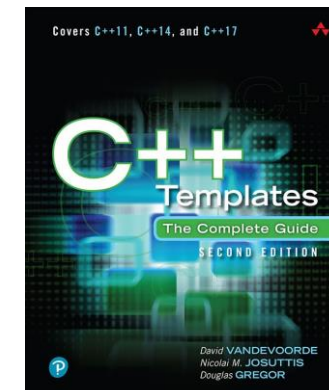
Vector_Dinamico_float.h

```
#include "Vector_Dinamico.h"
#include "Vector_Dinamico.hpp"

//Forzamos la instanciación de la clase Vector_Dinamico
//con el tipo base float, que se usa en el programa
template class      Vector_Dinamico<float>;
```

compilando por separado main.c, Vector_Dinamico_float.h y enlazándolos

Consultar *C++ Templates. The Complete Guide*



Abstracción en iteración

- Una de las aplicaciones más importantes de la abstracción por especificación es la definición de **abstracciones iterativas**
- El **iterador** facilita el acceso a los elementos de colecciones (estructuras de datos con varios elementos) ignorando los detalles de su implementación y de las operaciones propias de la colección

Abstracción en iteración

- Supongamos que queremos sumar una colección de enteros

```
int suma(int v[], int n) {  
    int s=0;  
    for(int i=0; i<n; i++) {  
        s += v[i];  
    }  
    return s;  
}
```

La colección se almacena
en un vector

```
int suma(lista l) {  
    int n;  
    if (lista_vacia(l))  
        return 0;  
    else  
        n = primer_elemento(l);  
    return n + suma (resto_elementos(l))  
}
```

La colección se almacena
en una lista

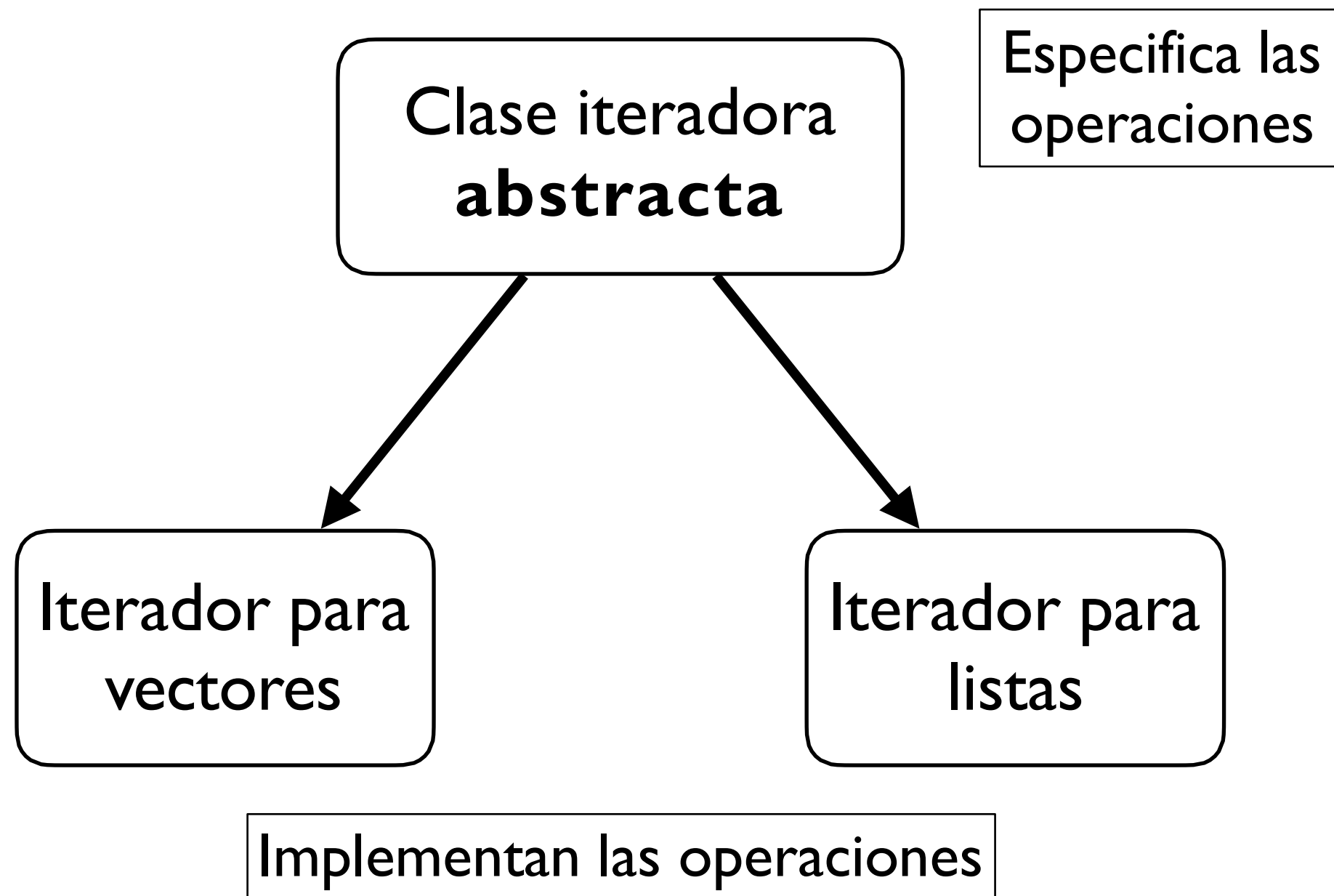
Abstracción en iteración

- Esquema general para el tratamiento de los elementos de una colección

```
obtener_primer_elemento();  
mientras (hay_elementos()) {  
    procesar (elemento_actual());  
    obtener_siguiente_elemento();  
}
```

- Donde:
 - › **obtener_primer_elemento()** inicializa el recorrido de la secuencia y, si no es vacía, deja disponible el primer elemento
 - › **hay_elementos()** indica si hay elementos no tratados (recorridos) en la secuencia
 - › **elemento_actual()** proporciona el elemento actual en la secuencia
 - › **obtener_siguiente_elemento()** deja disponible el siguiente elemento de la secuencia

Abstracción en iteración



Abstracción en iteración

- Podemos reescribir ahora el algoritmo así:

```
int suma(iterador itr){  
    int s=0;  
    obtener_primer_elemento(itr);  
    while(hay_elementos(itr))  
        s += elemento_actual(itr);  
        obtener_siguiente_elemento(itr);  
    }  
    return s;  
}
```

- Vemos que ahora no se accede directamente a las estructuras de datos.
- El algoritmo es independiente de las operaciones propias de la colección

Abstracción en iteración

- Programación genérica
- Poder recorrer cualquier contenedor (que admita ser recorrido) con cualquier tipo de dato base, siempre de la misma forma.

```
1  template <typename T>
2  void recorrer_elementos(const T& contenedor){
3      typename T::iterator iter;
4      for(iter=contenedor.begin();iter!=contenedor.end());++iter)
5          cout << *iter << endl;
6  }
```

Abstracción en iteración

- El iterador `iter` puede recorrer cualquier contenedor (vector, lista, conjunto,...) que se ajuste a esa especificación
- Ejemplo: Vector Dinámico con iteradores

TDAs en C++

- Los tipos predefinidos de C++ en realidad son TDAs.
- La idea es integrar en el lenguaje los nuevos tipos de datos que definamos de forma análoga a los predefinidos, de forma que códigos como éste sean válidos:

```
#include <iostream>
#include <polinomio.h>
using namespace std;
int main(){
    Polinomio p1, p2, resultado;
    cout<< "Introduzca el primer polinomio" << endl;
    cin >> p1;
    cout<< "Introduzca el segundo polinomio" << endl;
    cin >> p2;
    resultado = p1 + p2;
    cout <<"La suma de los polinomios es " << resultado << endl;
    return 0;
}
```

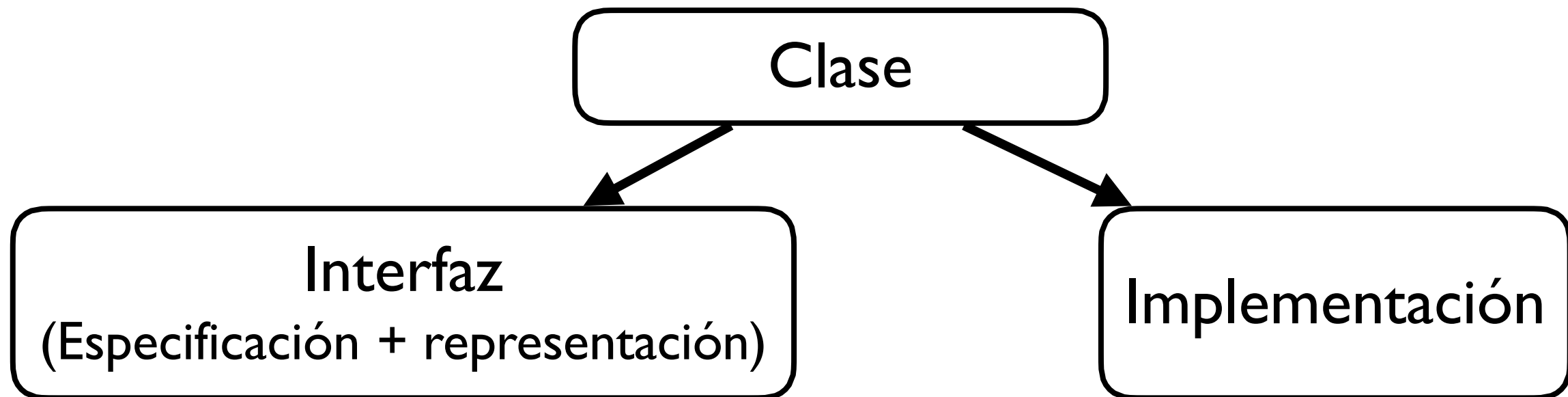
TDA en C++

- Recordemos, TDA = datos + operaciones
- Podemos implementar la unión de datos y operaciones mediante estructuras (struct)

```
struct Polinomio{  
    float * coef;           //Array de coeficientes  
    int grado;              //grado actual del polinomio  
    int MaxGrado;           //Máximo grado (espacio reservado)  
  
    void AsignarCoeficiente(int i, float c);  
    int Grado();  
    float Coeficiente(int i)  
    .....  
};
```

- Tiene inconvenientes, como la falta de ocultamiento. Por defecto, en las estructuras los miembros son públicos. También faltan otros mecanismos como la herencia...

TDAs en C++



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

- No podemos separar completamente la especificación de la implementación de la clase (la representación se incluye en la interfaz)

TDA en C++

- Una reflexión: ¿recuerdo lo estudiado en MP?
 - ▶ Control de acceso: private y public. Ámbito
 - ▶ Implementación de métodos miembro
 - ▶ Acceso a los datos miembro. El puntero this
 - ▶ Constructores. Destructor
 - ▶ Métodos inline
 - ▶ Métodos friend
 - ▶ Sobrecarga de operadores. Operadores como métodos miembro o métodos externos a la clase
 - ▶ El operador de asignación
 - ▶ Operadores de entrada/salida
 - ▶ Operadores unarios y binarios