

Objetivos del Tema

- ☒ 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 los mecanismos de herencia y composición.
- ☒ Ver cómo incorpora el lenguaje de programación a utilizar en las prácticas, el **C++**, éstos conceptos.

Abstracción en Programación

Abstracción: Operación intelectual que ignora selectivamente partes de un todo para facilitar su comprensión.

Abstracción en la resolución de problemas: Ignorar detalles específicos buscando generalidades que ofrezcan una perspectiva distinta, más favorable a su resolución.

Abstracción: descomposición en que se varía el nivel de detalle.

Propiedades de una descomposición útil:

- Todas las partes deben estar al mismo nivel
- Cada parte debe poder ser abordada por separado
- La solución de cada parte debe poder unirse al resto para obtener la solución final

• Abstracción en programación

```
encontrado ← falso;  
i ← límite_inferior(a);  
mientras i ≤ límite_superior(a) hacer  
    si a[i]=x entonces  
        z ← i; encontrado ← cierto  
        fsi;  
        i ← i+1  
fmientras
```

```
encontrado ← falso;  
i ← límite_superior(a);  
mientras i ≥ límite_inferior(a) hacer  
    si a[i]=x entonces  
        z ← i; encontrado ← cierto  
        fsi;  
        i ← i -1  
fmientras
```

```
encontrado ← esta_en(a,x);  
si encontrado entonces z ← indice_de(a,x) fsi
```

ABSTRACCION Y OLVITAMIENTO DE INFORMACION

CUANDO SE CONSTRUYE UN MODULO DIFERENCIAR:

* PARTE PUBLICA

Todos los aspectos que son visibles para los demás módulos.

** PARTE PRIVADA

Todos los detalles que no son visibles para los demás módulos



Proceso de Olvitamiento de Información

DOCUMENTACION

EL PROGRAMADOR DEBE CREAR 2 DOCUMENTOS DIFERENCIADOS:

* ESPECIFICACION

(.h)

- Características sintácticas y semánticas que describen la parte pública. Debe ser suficiente para usarlo y debe ser independiente de los detalles internos de construcción.

* IMPLEMENTACION

(.c(pp))

- Documento que presenta las características internas del módulo

Mecanismos de Abstracción en Programación

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

Ejemplo: cálculo de $\cos \alpha$.

- *Abstracción por especificación.* Permite abstraerse de la implementación concreta de un procedimiento asociándole una descripción precisa de su comportamiento.

Ejemplo: `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 procedimiento sin necesitar conocer otros elementos.

Abstracción por Especificación

Se suele expresar en términos de:

- **Precondición:** Condiciones necesarias y suficientes para que el procedimiento se comporte como se prevee.
- **Postcondición:** Enunciados que se suponen ciertos tras la ejecución del procedimiento, si se cumplió la precondición.

```
int busca_minimo(float * array, int num_elem)
/*
precondición:
    - num_elem > 0.
    - 'array' es un vector con 'num_elem'
        componentes.
postcondición:
    devuelve la posición del mínimo elemento
    de 'array'.
*/
```

Tipos de Abstracción

- **Abstracción Procedimental.** Definimos un conjunto de operaciones (procedimiento) que se comporta como una operación.
- **Abstracción de Datos (TDA).** Tenemos un conjunto de datos y un conjunto de operaciones que caracterizan el comportamiento del conjunto. Las operaciones están vinculadas a los datos del tipo.
- **Abstracción de Iteración.** Abstracción que permite trabajar sobre colecciones de objetos sin tener que preocuparse por la forma concreta en que se organizan.

Abstracción Procedimental

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

- La identidad de los datos no es relevante para el diseño. Sólo interesa el número de parámetros y su tipo.
- Con abstracción por especificación es irrelevante la implementación, pero no qué hace.
 - *Localidad*: Para implementar una abstracción procedural no es necesario conocer la implementación de otras que se usen, sólo su especificación.
 - *Modificabilidad*: Se puede cambiar la implementación de una abstracción procedural sin afectar a otras abstracciones que la usen, siempre y cuando no cambie la especificación.

Especificación de una Abs. Proc. ()

- a) **Cabecera:** (Parte sintáctica) Indica el nombre del procedimiento y el número, orden y tipo de las entradas y salidas. Se suele adoptar la sintaxis de un lenguaje de programación concreto. Ejemplo, el prototipo de la función en C++.
- b) **Cuerpo:** (Parte semántica) Está compuesto por las siguientes cláusulas.
 1. Argumentos (*Parámetros*)
 2. Requiere (*Precondiciones*)
 3. Valores de retorno
 4. Efecto
 5. *Excepciones*

Especificación de una Abs. Proc. (IV)

1. *Argumentos*: Explica el significado de cada parámetro de la abstracción, no el tipo, que ya se ha indicado en la cabecera, sino qué representa.
Indicar las restricciones sobre el conjunto datos del tipo sobre los cuales puede operar el procedimiento. Procedimiento *total* y *parcial*.
Indicar si cada argumento es modificado o no. Cuando un parámetro vaya a ser alterado, se incluirá la expresión “Es MODIFICADO”.
2. *Requiere*: Restricciones impuestas por el uso del procedimiento no recogidas en *Argumentos*. Ejemplo: que exista cierta operación para los objetos del tipo con que se llama a la abstracción o necesidad de que previamente se haya ejecutado otra abstracción procedural.

Especificación de una Abs. Proc. (V)

3. *Valores de retorno* Descripción de qué valores devuelve la abstracción y qué significan.
4. *Efecto*: Describe el comportamiento del procedimiento para las entradas no excluidas por los requisitos. El efecto debe indicar las salidas producidas y las modificaciones producidas sobre los argumentos marcados con “Es MODIFICADO”.

No se indica nada sobre el comportamiento del procedimiento cuando la entrada no satisface los requisitos. Tampoco se indica nada sobre cómo se lleva a cabo dicho efecto, salvo que esto sea requerido por la especificación. Por ejemplo, que haya un requisito expreso sobre la forma de hacerlo o sobre la eficiencia requerida.

5. *Excepciones*: Describe (si es necesario) el comportamiento de la función cuando se da una circunstancia que no permite la finalización con éxito de su ejecución.

EJEMPLOS

int IndiceMinimo (int vector[], int n elem)

/*

Precondiciones:

'n elem' > 0

'vector' es un vector de enteros con 'n elem' elementos

Postcondiciones:

• Devuelve la posición del mínimo elemento del vector

(devuelve $i \mid v[i] \leq v[j]$ para $0 \leq j < n elem$)

*/

int IndiceMaximo (int vec[], int n elem)

/*

Argumentos:

'vec': array 1-D que contiene los elementos donde encontrar el valor máximo

'n elem': número de elementos en el vector 'vec'

Devuelve:

índice en el vector 'vec'

Precondiciones:

'n elem' > 0

'vec' tiene al menos 'n elem' elementos

Efecto:

Busca el elemento más ~~pequeño~~^{pequeño} en el vector 'vec' y devuelve la posición de éste, es decir, el valor $i \mid v[i] \leq v[j]$ para $0 \leq j < n elem$

*/

Especificación junto al código fuente

- Falta de herramientas para mantener y soportar el uso de especificaciones \Rightarrow responsabilidad exclusiva del programador.
- Necesidad de vincular código y especificación.
- Incluir la entre comentarios en la parte de interfaz del código.
- Herramienta doc++. (<http://docpp.sourceforge.net>) Permite generar documentación de forma automática en distintos formatos (html, L^AT_EX, postscript, etc.) a partir del código fuente.
 - Herramienta : doxygen (<http://www.doxygen.org>)

Especificación usando doxygen

Toda la especificación se encierra entre:

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

1. Se pone una frase que describa toda la función: *Objetivo*
2. Cada argumento se precede de: @param.
3. Los requisitos adicionales se indican tras @pre ~~conditions~~.
4. Los valores de retorno se ponen tras @return.
5. La descripción del efecto sigue a @post.

ESPECIFICACION USANDO DOXYGEN

/*

- * @brief Calcula el indice del elemento maximo de un vector
- * @param vec array 1-D que contiene los elementos
 - * donde encontrar el valor maximo
- * @param nelem numero de elementos en el vector
 - * \a vec. \a nelem > 0
- * @pre \a vec es un vector de al menos \a nelem elementos
- * @return el indice del elemento mas grande en el vector \a vec, es decir, el valor \e i tal que
 - * \e vec[i] \geq \e vec[j] para \e 0 \leq j \leq nelem

*/

int indiceMaximo (int vec[], int nelem)

Abstracción de Datos

Tipo de Dato Abstracto (TDA): Entidad abstracta formada por un conjunto de datos y una colección de operaciones asociadas.

Ej.: tipos de datos en el lenguaje C++.

Conceptos manejados:

- *Especificación:* Descripción del comportamiento del TDA.
- *Representación:* Forma concreta en que se representan los datos en un lenguaje de programación para poder manipularlos.
- *Implementación:* La forma específica en que se expresan las operaciones.

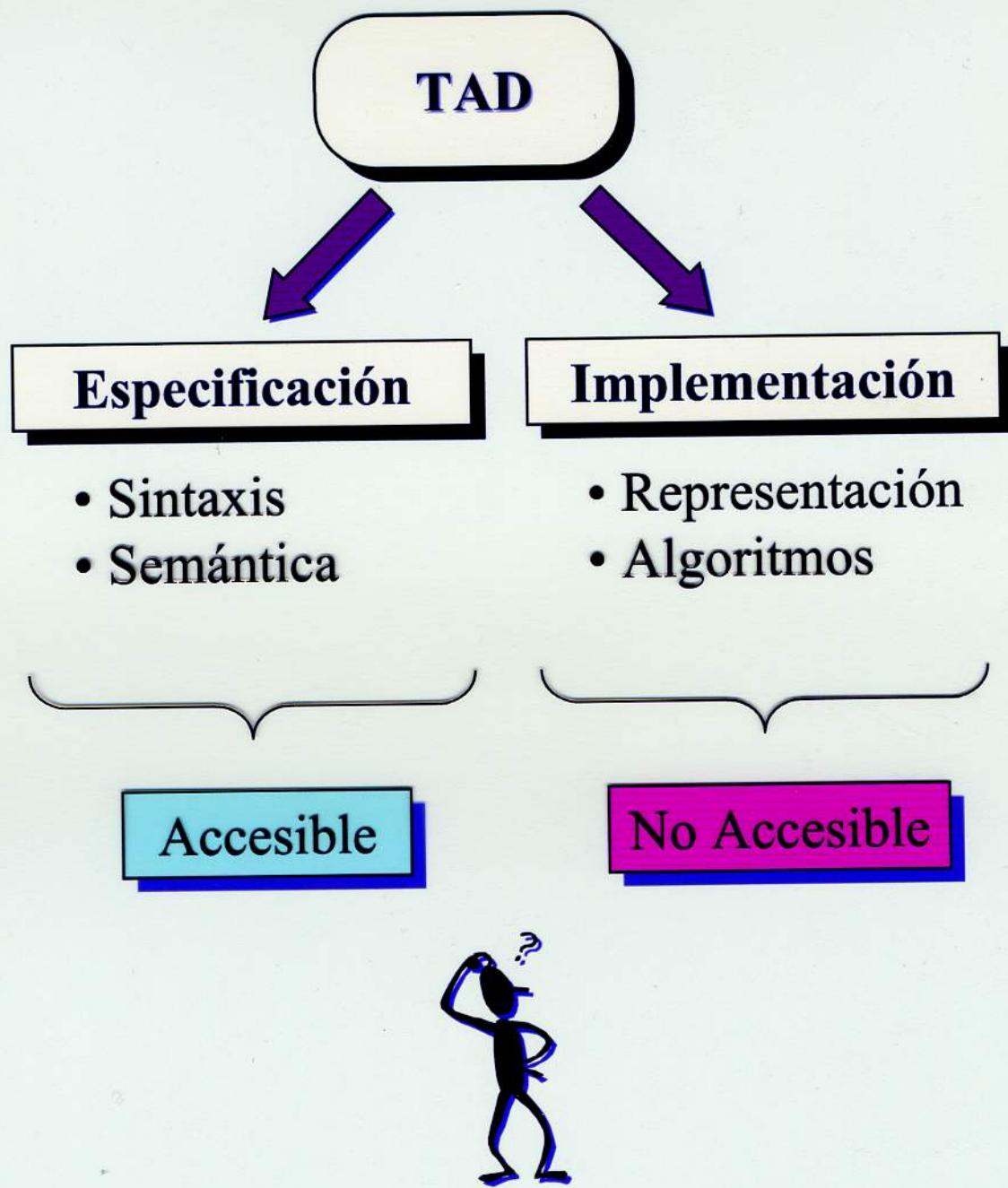
Visiones de un TDA

Hay dos visiones de un TDA:

- *Visión externa*: especificación.
- *Visión interna*: representación e implementación.

Ventajas de la separación:

- Se puede cambiar la visión interna sin afectar a la externa.
- Facilita la labor del programador permitiéndole concentrarse 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 modela, qué operaciones se les pueden aplicar, cómo se usan y qué hacen.

Estructura de la especificación:

1. *Cabecera*: nombre del tipo y listado de las operaciones.
2. *Definición*: Descripción del comportamiento sin indicar la representación. Se debe indicar si el tipo es mutable o no. También se expresa donde residen los objetos.
3. *Operaciones*: Especificar las operaciones una por una como abstracciones procedimentales.

TDA Fecha (I)

/**

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 (II)

```
void constructor (Fecha * f, int dia, int mes,  
                  int año)
```

```
/**
```

brief Constructor primitivo.

param f: Objeto creado. Debe ser nulo.

param dia: dia de la fecha. $0 < \text{dia} \leq 31$.

param mes: mes de la fecha. $0 < \text{mes} \leq 12$.

param año: año de la fecha.

Opc: 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.

```
*/
```

TDA Fecha (III)

```
void lee(Fecha * f, char * cadena)
```

```
/**
```

Objetivo Lee una fecha de una cadena.

Parametros f: Objeto creado. Es MODIFICADO.

Parametros cadena: Cadena de caracteres que representa una fecha en formato dd/mm/aaaa.

Return Objeto Fecha que representa la fecha leída en cadena.

```
*/
```

```
int dia(Fecha f)
```

```
/**
```

Objetivo Obtiene el día del objeto receptor.

Parametros f: Objeto receptor.

Return Dia del objeto f.

```
*/
```

TDA Fecha (IV)

```
int mes(Fecha f)
```

```
/**
```

~~Obrief~~ Obtiene el mes del objeto receptor.

 @param f: Objeto receptor.

 @return Mes del objeto f.

```
*/
```

```
int año(Fecha f)
```

```
/**
```

~~Obrief~~ Obtiene el año del objeto receptor.

 @param f: Objeto receptor.

 @return Año del objeto f.

```
*/
```

TDA Fecha (V)

```
char * Escribe (Fecha f, char * cadena)
```

```
/**
```

~~obnef~~ 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.

```
*/
```

TDA Fecha (VI)

```
void Siguiente(Fecha * f)
```

```
/**
```

~~abnef~~ Cambia f por la fecha siguiente a la que
representa.

```
@param f: Objeto receptor. Es MODIFICADO.
```

```
*/
```

```
void Anterior(Fecha * f)
```

```
/**
```

~~abnef~~ Cambia f por la fecha anterior a la que
representa.

```
@param f: Objeto receptor. Es MODIFICADO.
```

```
*/
```

TDA Fecha (VII)

```
bool menor(Fecha f1, Fecha f2)
```

```
/**
```

Objetivo Decide si f1 es anterior a f2.

Parametros f1, f2: Fechas que se comparan.

Return

true, si f1 es una fecha estrictamente anterior
a f2.

false, en otro caso.

```
*/
```

```
bool menor_o_igual(Fecha f1, Fecha f2)
```

```
/**
```

Objetivo Decide si f1 es anterior o igual que f2.

Parametros f1, f2: Fechas que se comparan.

Return

true, si f1 es una fecha anterior o igual a f2.
false, en otro caso.

```
*/
```

EJEMPLO DE USO DEL TDA FECHA

```
#include <iostream.h>
#include "fecha.h"

int main()
{
    Fecha f, g;
    int dia, mes, anio;
    char c1[100], c2[100];
    cout << "Introduzca el dia del mes: " << endl;
    cin >> dia;
    cout << "Introduzca el mes actual: " << endl;
    cin >> mes;
    cout << "Introduzca el año actual: " << endl;
    cin >> anio;
    constructor(&f, dia, mes, anio);
    siguiente(f, &g);
    if (menor(f, g))
    {
        Escribe(f, c1);
        Escribe(g, c2);
        cout << "El dia " << c1 << " es posterior a "
            << c2 << endl;
    }
    return 0;
}
```

Implementación de un TDA

Implica dos tareas:

- a) Diseñar una representación que se va a dar a los objetos.
- b) Basándose en la representación, implementar cada operación.

Dentro de la implementación habrá dos tipos de datos:

- a) Tipo Abstracto: definido en la especificación con operaciones y comportamiento definido.
- a) Tipo *rep*: tipo a usar para representar los objetos del tipo abstracto y sobre él implementar las operaciones.

Representación del TDA

- Tipo abstracto: TDA Fecha
- Tipo *rep*:

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

El tipo *rep* no está definido únicamente por el tipo abstracto: `typedef int Fecha[3];`

TDA Polinomio ($a_n x_n + \dots + a_1 x + a_0$).

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

Implementación de TDA

En toda implementación existen dos elementos característicos muy importantes:

- Función de abstracción: conecta los tipos abstracto y *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

Define el significado de un objeto *rep* de cara a representar un objeto abstracto. Establece una relación formal entre un objeto *rep* y un objeto abstracto.

$$f_A : rep \longrightarrow A$$

Es una aplicación sobreyectiva.

Ejemplos:

- TDA racional:

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

- TDA Fecha:

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

- TDA Polinomio:

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

Invariante de Representación

Invariante de Representación (I.R.): 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* $r=\{\text{num}, \text{den}\}$ debe cumplir: $\text{den} \neq 0$.
- TDA Fecha: Dado el objeto *rep* $f=\{\text{dia}, \text{mes}, \text{anio}\}$ debe cumplir:
 - $1 \leq \text{dia} \leq 31$
 - $1 \leq \text{mes} \leq 12$
 - Si $\text{mes} == 4, 6, 9 \text{ u } 11$, entonces $\text{dia} \leq 30$.
 - Si $\text{mes} == 2$ y bisiesto(anio) , entonces $\text{dia} \leq 29$.
 - Si $\text{mes} == 2$ y !bisiesto(anio) , entonces $\text{dia} \leq 28$.

Indicando la F.A. e I.R.

Tanto la función de abstracción como el invariante de la representación deben **FIGURAR ESCRITOS** en la implementación.

```
#include "racional.h"

/*
 ** Función de abstracción:
 -----
 fA : tipo_rep ----> q
 {num, den} ----> q
 La estructura {num, den} representa al número
 racional q = num/den.

 ** Invariante de Representación:
 -----
 Cualquier objeto del tipo_rep, {num, den},
 debe cumplir:
 - den != 0
 */
```

Preservación del I.R.

Es fundamental la conservación del I.R. para todos los objetos modificados por las operaciones que los manipulan. Su conservación se puede establecer demostrando que:

1. Los objetos creados por constructores lo verifican.
2. Las operaciones que modifican los objetos los dejan en un estado que verifica el I.R. antes de finalizar.

Sólo se podrá garantizar esto cuando exista ocultamiento de información.

TDA = COLECCION VALORES + OPERACIONES SOBRE ELLOS

↳ ESPECIFICACION + IMPLEMENTACION

• TIPO: DEFINICION → TIPO ABST.

• OPERACIONES

• TIPO → TIPO REP

• OPERACIONES

FD
IR

TIPOS: EJEMPLOS DE ESPECIFICACION E IMPLEMENTACION

TDA RACIONAL

Pareja de valores ~~enteros~~ (num, den) representando
un numero racional $\frac{\text{num}}{\text{den}}$ TIPO ABSTRACTO

Tipo rep

typedef int Rational [2]; or

{
typedef struct {
 int numerador;
 int denominador;
} Rational

F. A.

fabs: rep → Rational

r → $\frac{r.\text{numerador}}{r.\text{denominador}}$

I.R.

$r.\text{denominador} \neq 0$

TDA fecha

Representa fechas según el calendario occidental en el formato dd/mm/aaaa TIPO ABSTRACTO

Tipo Rep

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

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

```
typedef int Fecha;
```

F.A.

$f_{abs}: rep \rightarrow Fecha$

$r \rightarrow r.\text{dia}/r.\text{mes}/r.\text{anio}$

I.R.

Un objeto r que representa una fecha debe cumplir que:

- $1 \leq r.\text{dia} \leq 31$ y $1 \leq r.\text{mes} \leq 12$
- $r.\text{mes} \in \{4, 6, 9, 11\}$ $\rightarrow r.\text{dia} \leq 30$
- $r.\text{mes} = 2$ y $\text{bisiesto}(r.\text{anio}) \rightarrow r.\text{dia} \leq 29$
- $r.\text{mes} = 2$ y $\text{no bisiesto}(r.\text{anio}) \rightarrow r.\text{dia} \leq 28$

donde $\text{bisiesto}(i)$ es la función booleana que devuelve verdadero para los años bisiestos.

$$\text{bisiesto}(i) \equiv ((i \% 4 = 0) \text{ } \& \& \text{ } (i \% 100 \neq 0)) \text{ } || \text{ } (i \% 400 = 0)$$

TDA Polinomio

Sucesión de números reales $\{q_0, q_1, \dots, q_n\}$ representando el polinomio: $q_0 + q_1x + q_2x^2 + \dots + q_nx^n$ (grado n)

TIPO ABSTRACTO

Tipo Rep

```
{ struct poli {  
    float *coef;  
    int MaxGrado;  
    int grado;  
};
```

F. A.

fabs: rep \rightarrow Polinomio
 $P \rightarrow P \rightarrow \text{coef}[0] + P \rightarrow \text{coef}[1]x^1 + \dots +$
 $P \rightarrow \text{coef}[P \rightarrow \text{grado}]x^{P \rightarrow \text{grado}}$

I.R.

La condición para que un objeto representado sea válido es:

$$(P \neq 0, P \rightarrow \text{coef}[P \rightarrow \text{grado}] \neq 0)$$

iff

$$(P[i] = 0, \forall i / P \rightarrow \text{grado} < i \leq P \rightarrow \text{MaxGrado})$$

TDA Conjunto

Un conjunto es una colección de elementos de un tipo determinado (tipo base) que no pueden repetirse

TIPO ABSTRACTO

Tipo Rep

```
typedef int TipoBase;  
typedef struct {  
    TipoBase elementos[];  
    int nElem;  
} Conjunto;
```



F. A.

$fabs : \text{REP} \rightarrow \text{Conjunto}$
 $r \rightarrow \{ r.\text{elementos}[i] / 0 \leq i < r.\text{nElem} \}$

S. R.

$(r.\text{elementos}[i] \neq r.\text{elementos}[j]) \quad \forall i, j \quad 0 \leq i < j < r.\text{nElem}$

• Clases de operaciones

☞ Constructores primitivos

Crean objetos del tipo de dato sin tomar ningún objeto del mismo como entrada.

☞ Constructores

Crean objetos del tipo de dato tomando algún objeto del mismo como entrada.

☞ Modificadores

Modifican los objetos del tipo de dato.

☞ Observadores

Toman como entrada objetos del tipo de dato y retornan resultados de otros tipos.

☞ Destructores

Permiten eliminar un objeto del tipo de dato, recuperando el espacio de memoria de su representación.

Generalización o parametrización

Idea intuitiva

Subiendo en el nivel de abstracción: crear una abstracción que se pueda usar para más de un tipo de dato. Parametrizar el tipo de dato.

```
int minimo(int a, int b)
{
    return (a < b ? a : b);
}
```

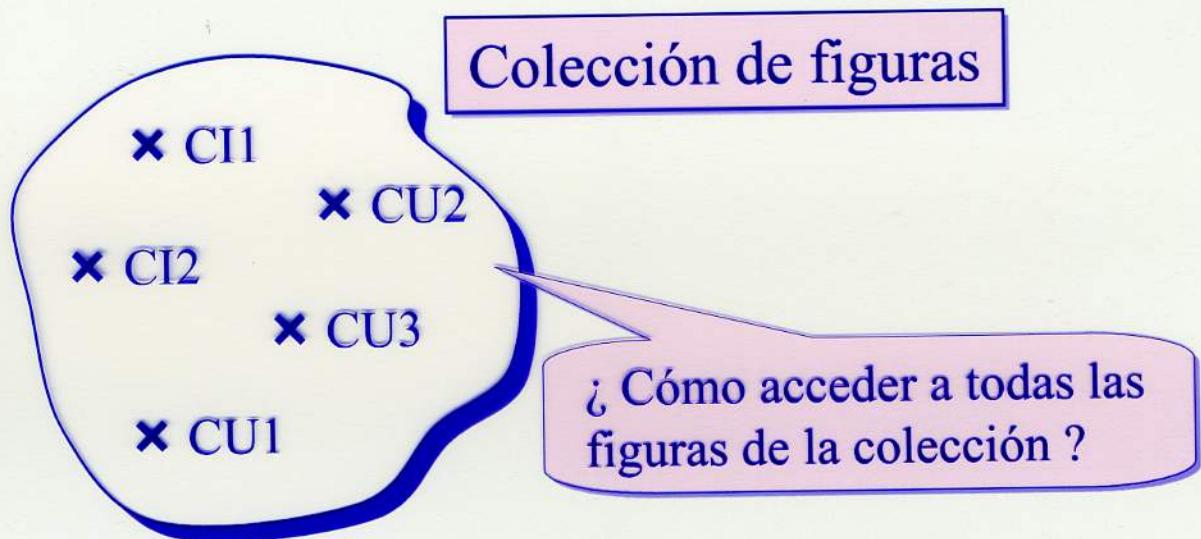
Esta función devuelve el mínimo de sus dos argumentos int. Sin embargo, la forma de calcular este valor no sólo es válida para cualquier par de int, sino también para cualquier par de valores para los que esté definida la operación <.

De modo más general:

```
T minimo(T a, T b)
{
    return (a < b ? a : b);
}
```

• Iteradores

Una de las aplicaciones más importantes de la abstracción por especificación es la definición de **abstracciones iterativas**.



Un **iterador** facilita el acceso a los elementos de colecciones (estructuras de datos con varios elementos) ignorando los detalles de implementación de las mismas y de las operaciones propias de la colección.

Inconveniente de no utilizar iteradores

Supongamos que queremos sumar una colección de $n \geq 0$ enteros.

```
función suma (v:vector) retorna entero;  
var s,i:entero;  
inicio  
    s ← 0; i ← 0;  
    mientras i≤n hacer  
        i ← i+1;  
        s ← s+v[i];  
    fmientras;  
    retorna s  
ffunción
```

La colección se almacena en un vector v.

La colección se almacena en una lista l.

¿ Por qué han de ser distintos ?



```
función suma (l:lista) retorna entero;  
var n:entero;  
inicio  
    si lista_vacía(l)  
        retorna 0  
    sino  
        n ← primer_elemento(l);  
        retorna n+suma(resto_elementos(l))  
    fsi  
ffunción
```

Esquema general de tratamiento de los elementos de una colección (secuencia)

```
obtener_primer_elemento;  
mientras hay_elemento? hacer  
    tratar(elemento_actual);  
    obtener_elemento_siguiente  
fmiéntras
```

Donde:

- **Obtener_primer_elemento.** Inicializa el recorrido de la secuencia y, si ésta no es vacía, deja disponible su primer elemento.
- **Hay_elemento?.** Indica si hay elementos en la secuencia no tratados (o recorridos).
- **Elemento_actual.** Proporciona el elemento actual (en curso) de la secuencia.
- **Obtener_elemento_siguiente.** Deja disponible el siguiente elemento de la secuencia.

Especifica las operaciones:

- `obtener_primer_elemento`
- `hay_elemento?`
- `elemento_actual`
- `obtener_elemento_siguiente`

Clase iteradora abstracta

Iterador para vectores

Iterador para listas

Implementan las operaciones

Ahora, el algoritmo para sumar enteros puede reescribirse como:

```
función suma (itr:iterador) retorna entero;  
var s,i:entero;  
inicio  
    s ← 0;  
    obtener_primer_elemento(itr);  
    mientras hay_elemento?(itr) hacer  
        s ← s+elemento_actual(itr);  
        obtener_siguiente_elemento(itr)  
    fmientras;  
    retorna s  
ffunción
```



Observese que ahora, no sólo no se accede a las estructuras de datos, sino que además el algoritmo es independiente de las operaciones propias de la colección.