

Tema 1

Diseño, implementación y uso de TADs

Estructura de datos

Facultad de Informática - Universidad Complutense de Madrid

Transparencias de los Profs.:

Mercedes Gómez Albarrán y José Luis Sierra Rodríguez



Abstracción

- Abstracción = Capacidad para separar o aislar la especificación de un proceso o artefacto (el qué) de sus detalles internos (el cómo, la implementación)
- Dos formas fundamentales de abstracción en los lenguajes de alto nivel
 - Abstracción procedimental (o funcional)
 - El artefacto es una operación
 - Reunir un conjunto de sentencias que realizan una operación sobre unos datos y “darles un nombre” y una interfaz que las abstraen
 - Se puede ver como una forma de añadir operaciones a un lenguaje
 - Es la base del diseño descendente
 - Abstracción de datos
 - El artefacto es un tipo de datos
 - Separar el comportamiento deseado de un tipo de datos de los detalles relativos a la representación y el manejo de los valores de ese tipo
 - Se puede ver como una forma de añadir tipos a un lenguaje
 - Es la base del diseño centrado en los tipos de datos



Tipos Abstractos de Datos

- Tipo de datos = grupo de valores + operaciones definidas sobre los valores
- Tipo Abstracto de Datos (TAD) = tipo de datos donde la representación de valores y la implementación de operaciones son “opacas”, están “ocultas” al usuario
 - La especificación de los valores y de las operaciones está separada de la representación de los valores y de la implementación de las operaciones
 - Un tipo de datos *a lo tipo de datos predefinidos*
 - Privacidad: la representación interna y la implementación de las operaciones está oculta y es invisible para los usuarios del TAD
 - Protección: el tipo sólo se puede usar a través de las operaciones definidas, los accesos incontrolados se imposibilitan
- Ejemplos de TADs predefinidos: bool, char, int y derivados, float, double, string,...



Tipos Abstractos de Datos vs. Estructuras de datos

- Una estructura de datos es una estrategia de almacenamiento en memoria de la información que se desea guardar
 - La lista enlazada y los arrays son dos estructuras de datos tradicionales para implementar los TADs lista, pila y cola
- La importancia de estudiar las estructuras de datos radica en conocer aspectos como los costes de las operaciones que proporcionan, que es imprescindible para tomar decisiones acerca de cuándo decantarse por una u otra estructura



Algunos motivos por los que usar TADs

- Simplifican el desarrollo de software
 - Escenario: en un proyecto software se han identificado 3 TADs a usar
 - Los miembros del equipo de desarrollo encargados del desarrollo de cada TAD y los encargados del código “pegamento” no interfieren, siempre respetando y asumiendo las especificaciones acordadas para los tipos
 - Si hay un único miembro: la carga cognitiva se reduce pues se centra en una cosa cada vez
- Favorecen la reutilización
 - Un mismo TAD puede usarse múltiples veces
- El código que usa TADs es más compacto y legible que uno equivalente que integre *todo en uno*
- Las modificaciones y mejoras en la representación e implementación del tipo son transparentes para el usuario, no afectan a la *interfaz* del tipo ni a los programas desarrollados o en proceso de desarrollo que usen el tipo

(Horowitz et al. 1995, pp. 8-11)



Algunos conceptos relacionados con los TADs

- Tipos representantes = tipos de implementación concretos que se van a usar para representar el TAD
- Función de abstracción = correspondencia entre la representación y los valores del TAD, interpretación que se va a hacer de los valores de los tipos representantes del TAD
- Invariante de la representación = condiciones que deben cumplirse para que una representación sea válida
 - Ej.: TAD Rectángulo representado por un punto del plano, su anchura y su altura
 - La anchura y la altura deben ser ≥ 0 (si $0 \rightarrow$ rectángulo vacío)
 - Ej.: TAD Rectángulo representado por dos puntos que sean esquinas opuestas
 - Las coordenadas x e y del segundo punto mayores o iguales que las del primero (si iguales \rightarrow rectángulo vacío)



Algunos conceptos relacionados con los TADs

- Tipos de operaciones de los TADs
 - Constructoras (o generadoras) = crean los valores del tipo
 - Observadoras = devuelven información/aspecto de los valores del tipo que pertenece a otros tipos de datos ya conocidos
 - Mutadoras (o modificadoras) = modifican un valor del tipo, su resultado es un valor del tipo
 - Destructoras = establecen las condiciones necesarias para que un valor del tipo pueda “desaparecer” (liberación de cualquier recurso reservado para su creación)

Algunos conceptos relacionados con los TADs

- Para un mismo TAD podemos elegir distintas constructoras
- Según las constructoras elegidas puede que haya una única forma de obtener y representar con ellas cada valor del TAD o que haya diferentes representaciones para un mismo valor del TAD
 - Relación (o función) de equivalencia: indica cuándo dos representaciones (dos valores de los tipos representantes) distintas corresponden al mismo valor del TAD
 - Ejemplo: constructoras para el TAD número entero
 - cero: entero \rightarrow entero
 - suc: entero \rightarrow entero
 - pred: entero \rightarrow entero

suc(pred(suc(pred(cero)))) y cero son términos equivalentes

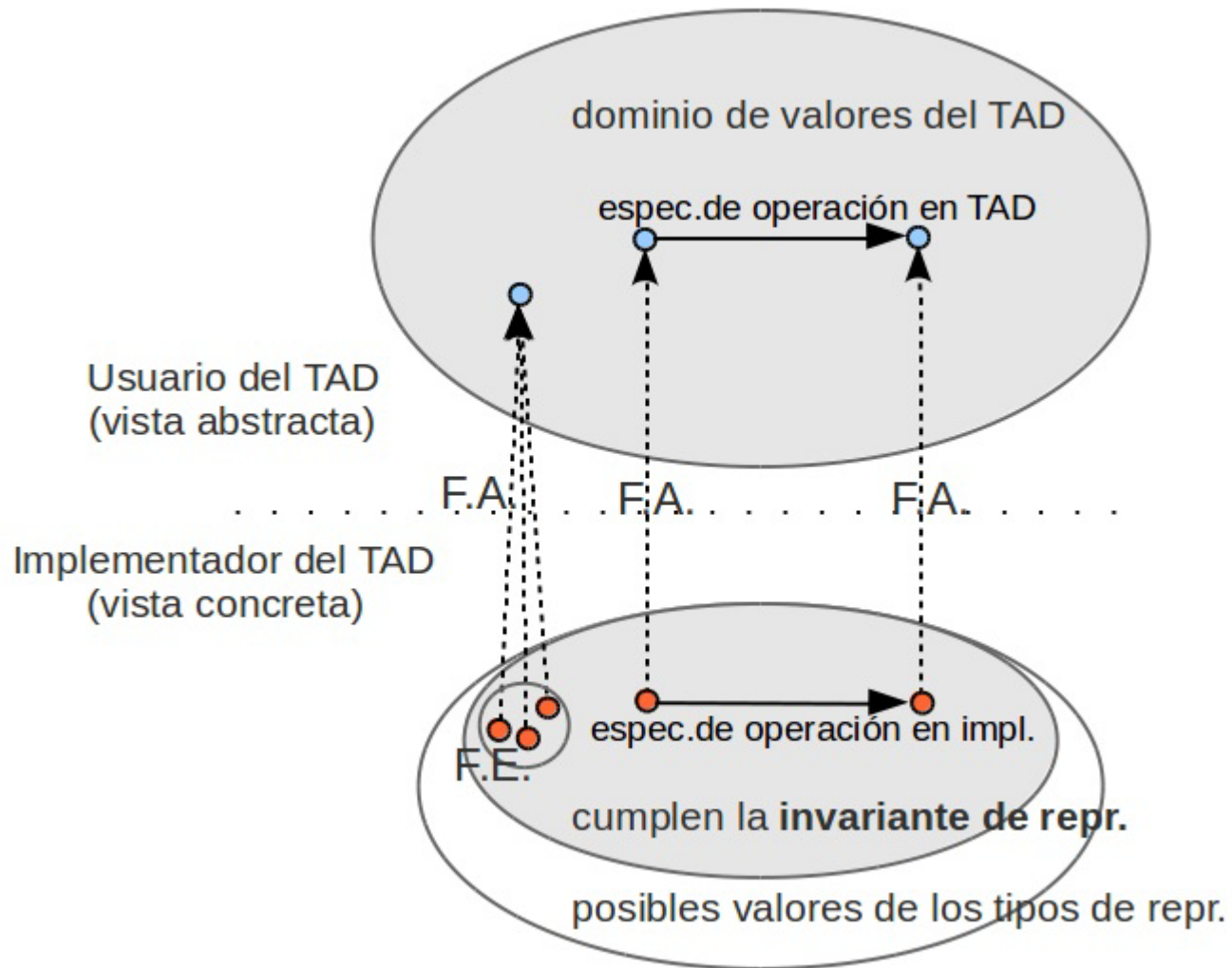
Ecuaciones de equivalencia: $\text{suc}(\text{pred}(n)) = n$ $\text{pred}(\text{suc}(n)) = n$



Algunos conceptos relacionados con los TADs

- Parcialidad de las operaciones de un TAD
 - Una operación (constructora, observadora, mutadora) es parcial cuando hay situaciones en las que no está definida y conducen a situaciones de error
 - Provocadas por limitaciones impuestas por recursos finitos (memoria, ficheros) o debidas a los tipos representantes seleccionados (vector de tamaño fijo)
 - Otras veces las operaciones son por definición erróneas (intentar dividir por cero, intentar acceder fuera de un vector, etc.)
 - Cualquier operación parcial conlleva precondiciones adicionales que garanticen un comportamiento predecible, y debe estar debidamente comentada
 - Necesidad de manejo de errores

Algunos conceptos relacionados con los TADs





Definición e implementación de un TAD

- Pasos para implementar un TAD:
 1. Elegir los tipos de implementación concretos que se va a usar para representar el TAD (tipos representantes)
 2. Dar la interpretación que se va a hacer de sus valores (función de abstracción)
 3. Definir las condiciones que se deben cumplir para que los valores representantes se consideren validos (invariantes de representación)
 4. Decidir cuando dos valores del tipo representante corresponden a un mismo valor del TAD (función de equivalencia)
 5. Implementar las operaciones especificadas, de forma que nunca se rompan los invariantes de representación (posiblemente con restricciones adicionales debidas al tipo representante elegido)



Definición e implementación de un TAD

- Elección de tipos representantes: un TAD se puede implementar de diversas formas
 - TAD Rectángulo alineado con los ejes representado por un punto del plano, su anchura y su altura *vs.* TAD Rectángulo representado por dos puntos que sean esquinas opuestas
 - TAD Numero_complejo representado por una parte real y una parte imaginaria *vs.* TAD Numero_complejo en representación polar (módulo-argumento)
 - TAD Fecha representado por día-mes-año *vs.* TAD Fecha representado por n° de días transcurridos desde 1-1-1900 *vs.* TAD Fecha representado por n° de segundos transcurridos desde las 00:00:00 horas del 1/1/1970
 - TAD Lista representado mediante array con contador *vs.* TAD Lista representado mediante lista enlazada
 - TAD Conjunto representado mediante array con contador *vs.* TAD Conjunto representado mediante lista enlazada
- Cada implementación tendrá unas limitaciones diferentes y un coste diferente para sus operaciones
 - Representación de la fecha dd-mm-aa (año mediante dos dígitos) y efecto Y2K
 - Representación de la fecha con segundos desde 1/1/1970 (entero sin signo de 32 bits) y problema del año 2038
 - Coste $O(n)$ de la observadora elemento_íésimo de una lista soportada por lista enlazada



Definición e implementación de un TAD

- TAD Rectángulo alineado con los ejes.- Primera representación
 1. Tipos representantes: Un Punto y un par de n° reales.
 2. Función de abstracción: El Punto representa el extremo inferior izquierdo del rectángulo. Los valores reales: el ancho y el alto. Un rectángulo es vacío si $\text{alto} = \text{ancho} = 0$.
 3. Invariante de la representación: El ancho y alto deben ser valores mayores o iguales que cero.
 4. Función de equivalencia: Dos rectángulos son iguales cuando tiene el mismo origen y el mismo alto y ancho. Los rectángulos vacíos se consideran iguales.



Definición e implementación de un TAD

- TAD Rectángulo alineado con los ejes.- Segunda representación
 1. Tipos representantes: Dos puntos
 2. Función de abstracción: Los puntos representan la esquina inferior izquierda y la esquina superior derecha del rectángulo. Un rectángulo es vacío si sus dos puntos son iguales.
 3. Invariante de la representación: Las coordenadas del segundo punto deben ser mayores (o iguales) que las del primero.
 4. Función de equivalencia: Dos rectángulos son iguales cuando sus puntos coinciden.



Definición e implementación de un TAD

- TAD Fecha.- Primera representación
 1. Tipos representantes: Tres enteros.
 2. Función de abstracción: Los enteros representan el día, el mes y el año.
 3. Invariante de la representación: Se deben respetar las reglas de construcción de fechas válidas (meses con 28, 29, 30 o 31 días, meses del 1 al 12, etc.)
 4. Función de equivalencia: Dos fechas son iguales si el día, el mes y el año de una coinciden con los de la otra.



Definición e implementación de un TAD

- TAD Fecha.- Segunda representación
 1. Tipos representantes: Un entero.
 2. Función de abstracción: El entero representa los días transcurridos desde el 1 de enero de 1900 (positivo si fecha posterior; negativo si fecha anterior).
 3. Invariante de la representación: Válido cualquier valor positivo o negativo.
 4. Función de equivalencia: Dos fechas son iguales cuando su valor representante coincide.

Definición e implementación de un TAD

■ TAD Conjunto.- Primera representación

1. Tipos representantes: Un array estático `_elementos` y un entero `_tam` que hace de contador de elementos en el array.
2. Función de abstracción: Los elementos del array con índices 0 a `_tam-1` forman parte del conjunto (salvo posibles repeticiones)

3. Invariante de la representación: distintas alternativas

- I1: El contador de elementos $\in [0, \text{tamaño_array}]$

`_tam=3` `_elementos`

2	1	2	...
---	---	---	-----

- I2: El contador de elementos $\in [0, \text{tamaño_array}]$ y los elementos del array entre 0 y `_tam-1` no están repetidos

`_tam=3` `_elementos`

2	1	3	...
---	---	---	-----

- I3: El contador de elementos $\in [0, \text{tamaño_array}]$, los elementos del array entre 0 y `_tam-1` no están repetidos y además están ordenados (supuesto que existe una relación de orden entre ellos)

`_tam=3` `_elementos`

1	2	3	...
---	---	---	-----

Definición e implementación de un TAD

- TAD Conjunto.- Primera representación (cont.)

- 4. Función de equivalencia

- Con I1: las repeticiones y el orden de los elementos son irrelevantes

_tam=2 _elementos

2	1
---	---	-----	-----

_tam=3 _elementos

1	1	2	...
---	---	---	-----

- Con I2: el orden de los elementos es irrelevante

_tam=3 _elementos

2	1	3	...
---	---	---	-----

_tam=3 _elementos

3	1	2	...
---	---	---	-----

- Con I3: vectores equivalentes sólo difieren en la parte "vacía" del array

_tam=3 _elementos

1	2	3	...
---	---	---	-----

_tam=3 _elementos

1	2	3	...
---	---	---	-----

Constructoras de tipos disponibles en los lenguajes de programación: ¿herramientas de implementación de TADs?

- Las constructoras de tipos (como struct de C) no permiten ocultar los detalles al usuario

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

- Libre acceso a la representación interna del tipo de datos
- Posibilidad de crear fechas inconsistentes de acuerdo con la semántica pretendida

```
Fecha f;  
f.mes = 2;  
f.dia = 30;
```

Constructoras de tipos disponibles en los lenguajes de programación: ¿herramientas de implementación de TADs?

- Junto con la implementación del concepto de fecha anterior se pueden crear subprogramas que manipulen variables de ese tipo

```
struct Fecha {  
    int dia; int mes; int anyo;  
};  
void inicia_fecha(Fecha& f, int d, int m, int a);  
void incrementa_anyo(Fecha& f, int n);  
// otras
```

- No implica que esas funciones sean las únicas que dependan directamente de la representación de Fecha
- Ni son las únicas que pueden acceder a la representación interna de una fecha

(En el libro "El lenguaje de programación C++" de Stroustrup -sección 10.2- se detallan más las posibilidades que ofrece struct en C++)



Las clases: herramienta de implementación de TADs

- Para que un lenguaje de programación soporte la implementación de TADs, el lenguaje debe contar con mecanismos que permitan separar la especificación de la implementación (cada lenguaje de programación puede ofrecer diferentes niveles de “aislamiento”):
 - Protección, de forma que los valores del tipo sólo puedan usarse por medio de las operaciones que aparecen en la especificación. Los intentos de acceso a partes privadas producen errores de compilación (ej.: clases Java o C++)
 - Mayor privacidad si además se usan clases abstractas puras
 - Convención, “pacto entre caballeros” para no “tocar” lo marcado como privado (una marca típica es comenzar los identificadores privados con `_`; p.e.: `_tam`)
- Basta con clases concretas (no abstractas puras)
 - Conocer “el interior” del tipo no es un problema si sólo pueden manipularse sus valores a través de las operaciones de la especificación
- Las clases de los lenguajes OO están pensadas para permitir que los tipos definidos por el programador reciban el mismo tratamiento que los tipos predefinidos

Las clases de C++ son una herramienta para implementar TADs

Las clases: herramienta de implementación de TADs

- TAD Fecha (primera representación: tres enteros)

```
class Fecha {
public: // parte publica del TAD
    // Constructor
    Fecha(int dia, int mes, int anyo);
    // Constructor alternativo: dias es un número de días a sumar/restar a la fecha base
    Fecha(const Fecha &base, int dias);
    // devuelve el dia de esta fecha (Observadora)
    int dia() const;
    // devuelve el mes de esta fecha (Observadora)
    int mes() const;
    // devuelve el año de esta fecha (Observadora)
    int anyo() const;
    // distancia, en días, de la fecha actual con la dada (Obs.)
    int distancia(const Fecha &otra) const;
    // devuelve el día de la semana de esta fecha (Observadora)
    int diaSemana() const;

private: // parte privada, accesible solo para la implementación
    int _dia;           // entre 1 y 28,29,30,31, en función de mes y año
    int _mes;           // entre 1 y 12, ambos inclusive
    int _anyo;

};
```



Uso de const para observadoras y en algunos parámetros



Convención -> comenzar identificadores privados por _

Las clases: herramienta de implementación de TADs

```
class Fecha {
public: // parte publica del TAD
    // ...
    Fecha(int dia, int mes, int anyo);
    int dia() const;
    int mes() const {return _mes;}
    int anyo() const {return _anyo;}
    // ...
private: // parte privada, accesible solo para la implementación
    int _dia;           // entre 1 y 28,29,30,31, en función de mes y año
    int _mes;          // entre 1 y 12, ambos inclusive
    int _anyo;
};
Fecha::Fecha(int dia, int mes, int anyo) {
    _dia = dia; _mes = mes; _anyo = anyo;
}
int Fecha::dia() const { return _dia; }

int main() {
    Fecha f(27, 3, 2014);
    cout << "el día es = " << f.dia() << "\n";
    cout << "el mes es = " << f.mes() << "\n";
    cout << "el año es = " << f.anyo() << "\n";
    return 0;
}
```

Implementación *inline* e implementación fuera de la clase

Aún no controlamos errores en la creación de fechas

Uso de los valores del tipo

Las clases: herramienta de implementación de TADs

Por defecto todos los miembros de una clase son privados a no ser que se especifique lo contrario

```
class Fecha {  
    public: // parte publica del TAD  
        Fecha(int dia, int mes, int anyo);  
        int dia() const;  
        // ... resto de operaciones  
    private:  
        int _dia;  
        int _mes;  
        int _anyo;  
};
```



```
class Fecha {  
    int _dia;  
    int _mes;  
    int _anyo;  
    public: // parte publica del TAD  
        Fecha(int dia, int mes, int anyo);  
        int dia() const;  
        // ... resto de operaciones  
};
```





Las clases: herramienta de implementación de TADs

- TAD Conjunto de enteros (primera representación)
 - Representación: array + contador
 - Invariante de la representación: $I1 = \text{El contador} \in [0, \text{tamaño_array}]$
 - La elección del invariante tiene impacto en la forma de implementar las operaciones del tipo, y por tanto en el coste de las mismas
 - Estrategia de manejo de errores de las operaciones parciales: devolución de valor de error (manejo de errores *a lo FP*)
 - Requiere que haya un valor de error disponible que no pueda ser confundido con una respuesta de no-error o devolver un booleano indicando el estado de error
 - El código cliente debe verificar mediante algún condicional si ha habido o no error

Las clases: herramienta de implementación de TADs

```
class Conjunto {
    static const int MAX = 50;
    int _tam;
    int _elementos[MAX];
public:
    // Constructor
    Conjunto();
    // inserta un elemento (mutadora)
    // parcial: si se intenta insertar más de MAX elementos, error (debido a la
    // limitación de tamaño del tipo representante). Devuelve si hay error o no.
    bool inclusion(int e);
    // elimina un elemento (mutadora)
    // parcial: si e no pertenece, error. Devuelve si hay error o no.
    bool exclusion(int e);
    // si el elemento pertenece al conjunto, devuelve 'true' (observadora)
    bool pertenencia(int e) const;
    // indica si el conjunto está vacío (observadora)
    bool esVacio() const;
    // otras operaciones (p.e. operaciones de combinación de
    // conjuntos: unión, intersección, diferencia)
};
```

 static → atributo de clase
(compartido por todo objeto
de la clase)

Las clases: herramienta de implementación de TADs

```
Conjunto::Conjunto() { _tam = 0; }  
// O(1)
```

```
bool Conjunto::inclusion(int e) {  
    bool error = false;  
    if (_tam == MAX) error = true;  
    else {  
        _elementos[_tam] = e;  
        _tam ++;  
    }  
    return error;  
}  
// O(1)
```

```
bool Conjunto::exclusion(int e) {  
    bool esta = false;  
    int i=0;  
    while (i<_tam)  
        if (_elementos[i]==e){  
            // pasamos el ultimo elemento a su lugar  
            _elementos[i] =  
                _elementos[_tam-1];  
            _tam --;  
            esta = true;  
        }  
        else i++;  
    return !esta;  
}  
// O(_tam)
```

Las clases: herramienta de implementación de TADs

```
bool Conjunto::pertenencia(int e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta){
        esta = _elementos[i]==e;
        i++;
    };
    return esta;
}
// O(_tam)
```

```
bool Conjunto::esVacio() const{
    return _tam == 0;
}
// O(1)
```

Implementación modular de los TADs

- *Declaración* de la clase en el archivo .h (fecha.h en el ejemplo)

```
// includes imprescindibles para que compile el .h
#include <libreria_sistema>
#include "modulo_propio.h"
// protección contra inclusión múltiple
#ifndef _FECHA_H_
#define _FECHA_H_
// ... declaración de la clase Fecha aquí...
// fin de la protección contra inclusión múltiple
#endif
```



Documentación del módulo:
en el .h

- Implementación de la clase en el .cpp asociado al .h (fecha.cpp)

```
#include "fecha.h"
// includes adicionales para que compile el .cpp
#include <libreria_sistema>
#include "modulo_propio.h"
// implementación de las operaciones de Fecha ...
```

Clientelismo entre TADs: Uso de TADs en TADs

```
/* Un punto del plano */
class Punto {
    float _x, _y;
public:
    // cambian las coordenadas del punto
    void pon_x(float abcisa);
    void pon_y(float ordenada);
    // devuelven las coordenadas del punto (Obs.)
    float dame_x() const;
    float dame_y() const;
    // pinta un punto (Obs.)
    void dibujate() const;
    // equivalencia de puntos mediante p1 == p2 (Obs.)
    bool operator==(const Punto &p) const;
    // ... resto operaciones (constructores, etc.)
};
```

Punto.h



La relación de equivalencia es abstracta y en C++ se traslada a la implementación sobrecargando el operador ==

Clientelismo entre TADs: Uso de TADs en TADs



La clase Rectangulo es cliente de la clase Punto

```
#include "Punto.h" // usa Punto
/*
 * Un Rectangulo en 2D alineado con los ejes.
 */
class Rectangulo {
    Punto _origen;
    float _ancho, _alto; // Representación 2
public:
    // origen es la esquina inf. izquierda
    Rectangulo(Punto origen, float alto, float ancho);
    // interpreta ambos puntos como esquinas opuestas
    Rectangulo(Punto uno, Punto otro);
    // devuelve el punto con coordenadas x e y mínimas (Obs.)
    const Punto &origen() const;
    // devuelven las dimensiones del rectangulo (Obs.)
    float alto() const;
    float ancho() const;
```



Convenio: Si el valor devuelto por una observadora es *grande* y forma parte de un objeto existente (ej. receptor) devolveremos una referencia (mayor eficiencia, se evita realizar una copia) constante (para evitar que modificaciones sobre ese valor afecten al objeto con el que comparte)

Rectangulo.h

Clientelismo entre TADs: Uso de TADs en TADs

```
// equivalencia (Obs.) todos los rectangulos vacios son iguales
bool operator==(const Rectangulo &r) const;
// calcula area (Obs.)
float area() const;
// verifica si esta vacio, es decir, si tiene área 0 (Obs.)
bool esVacio() const;
// devuelve true si el punto está dentro (Obs.)
// el borde se considera dentro, excepto si el rectangulo está vacio
bool dentro(const Punto &p) const;
// indica si este rectangulo contiene a r (Obs.)
// true si todos los puntos de r están dentro
bool dentro(const Rectangulo &r) const;
// calcula interseccion (Obs.)
// devuelve el rectangulo con todos los puntos dentro de ambos
Rectangulo interseccion(const Rectangulo &r) const;
// dibuja el rectangulo (Obs.)
void dibujate() const;
```

```
};
```



Observar la diferencia cuando el valor devuelto es *grande* pero es un objeto nuevo construido en el método



Clientelismo entre TADs: Uso de TADs en TADs


// Implementación de la relación de equivalencia entre rectángulos

```
bool Rectangulo::operator==(const Rectangulo &r) const {  
    return (esVacio() && r.esVacio()) ||  
           (_alto == r._alto && _ancho == r._ancho  
            && _origen == r._origen);  
};
```

Rectangulo.cpp

Clientelismo entre TADs: Uso de TADs en TADs

```
#include "Punto.h" // usa Punto
/*
 * Un circulo
 */
class Circulo {
    Punto _centro;
    float _radio;
public:
    // cambian el centro y el radio del circulo
    void pon_centro(Punto p);
    void pon_radio(float f);
    // obtienen el centro y el radio del circulo (Obs.)
    const Punto& dame_centro() const;
    float dame_radio() const;
```

 La clase Circulo es cliente de la clase Punto

Circulo.h



Clientelismo entre TADs: Uso de TADs en TADs

```
// calcula longitud de la circunferencia (Obs.)  
float circunferencia() const;  
// calcula area (Obs.)  
float area() const;  
// equivalencia (Obs.)  
bool operator==(const Circulo &c) const;  
// dibuja el circulo (Obs.)  
void dibujate() const;  
// ... resto operaciones  
};
```



Implementaciones estáticas y dinámicas de TADs

- A veces los valores de los TADs pueden llegar a requerir mucho espacio; o su tamaño puede variar mucho en una ejecución de la aplicación, de ejecución en ejecución de una aplicación o según la aplicación en la que se reuse
- En esos casos conviene realizar implementación dinámica del TAD
- Memoria dinámica:
 - Se gestiona mediante punteros
 - Reserva y liberación de memoria mediante los operadores `new` y `delete` (o sus variantes `new[]` y `delete[]` si se trata de reservar y liberar arrays)



Implementaciones estáticas y dinámicas de TADs

- Errores comunes en el uso de punteros y memoria dinámica:
 - Usar un puntero sin haberle asignado memoria (mediante un `new` / `new[]` previo)
 - No liberar un puntero tras haber acabado de usarlo (mediante un `delete` / `delete[]`) → fuga de memoria (*memory leak*)
 - Intentar liberar varias veces la memoria apuntada por un puntero
 - Acceder a memoria ya liberada
- Para evitar estos errores, se recomiendan las siguientes prácticas:
 - Inicializar los punteros nada más declararlos. Si no es posible inicializarlos inmediatamente, conviene asignar el valor NULL (o el valor `nullptr` C++11)
 - Los accesos indebidos producirán errores inmediatos y más fáciles de diagnosticar
 - Nada más liberar un puntero, asignar el valor NULL (o `nullptr`)
 - `delete()` nunca libera punteros a NULL/`nullptr` con lo que se pueden evitar algunas liberaciones múltiples



Implementaciones estáticas y dinámicas de TADs

- En las implementaciones dinámicas de los TADs cobran vital importancia los siguientes métodos de la clase
 - Los constructores
 - Como parte de las inicializaciones, se harán los news
 - Los destructores
 - Como parte de la liberación de recursos se harán los deletes necesarios
 - El constructor de copia y el operador de asignación
 - Las versiones por defecto hacen copia bit a bit y... ¡puede no ser eso lo que se quiere!

Implementaciones estáticas y dinámicas de TADs

- Implementación dinámica de los TAD Punto y Circulo

```
class Punto {  
    float * _x, * _y;  
public:  
    // constructor  
    Punto(float xx = 0, float yy = 0){  
        _x = new float;      *_x = xx;  
        _y = new float;      *_y = yy;  
    }  
    // destructor  
    ~Punto() { delete _x; delete _y; }
```

Implementaciones estáticas y dinámicas de TADs

```
// constructor de copia
Punto (const Punto& p) {
    _x = new float;      *_x = *p._x;
    _y = new float;      *_y = *p._y;
}

// sobrecarga del operador de asignación: copia los contenidos
// (apuntados por atributos)
Punto& operator= (const Punto& p) {
    *_x = *p._x;
    *_y = *p._y;
    return *this;
}

//... resto de operaciones
};
```


Implementaciones estáticas y dinámicas de TADs

```
#include "Punto.h"
int main(){
    Punto p1;
    Punto p2(10,20);
    return 0;
}
```



¿Qué métodos se ejecutan?

Implementaciones estáticas y dinámicas de TADs

```
#include "Punto.h"
int main(){
    Punto p1;
    Punto p2(10,20);
    p1 = p2;
    return 0;
}
```



¿Qué métodos se ejecutan?

Implementaciones estáticas y dinámicas de TADs

```
#include "Punto.h"
int main(){
    Punto p1;
    Punto p3(p1);
    return 0;
}
```



¿Qué métodos se ejecutan?

Implementaciones estáticas y dinámicas de TADs

```
#include "Punto.h"
int main(){
    Punto p1;
    Punto p3 = p1;
    return 0;
}
```



¿Qué métodos se ejecutan?

Implementaciones estáticas y dinámicas de TADs

```
#include "Punto.h"
int main(){
    Punto* p;
    p = new Punto;
    p->dibujate();
    delete p;
    return 0;
}
```



Actúan el constructor y el destructor de Punto



Implementaciones estáticas y dinámicas de TADs

```
class Circulo {
    Punto* _centro;
    float* _radio;
public:
    // constructor
    Circulo(Punto p, float r=1) {
        _radio = new float;    *_radio = r;
        _centro = new Punto;  *_centro = p;
    }
    // destructor
    ~Circulo() {
        delete _centro;
        delete _radio;
    }
}
```



Implementaciones estáticas y dinámicas de TADs

```
// constructor de copia
Circulo (const Circulo& c) {
    _radio = new float;    *_radio = *c._radio;
    _centro = new Punto;  *_centro = *c._centro;
}

// sobrecarga del operador de asignación: copia los contenidos
// (apuntados por atributos)
Circulo& operator= (const Circulo& c) {
    *_radio = *c._radio;
    *_centro = *c._centro;
    return *this;
}

// ... resto operaciones
};
```

Implementaciones estáticas y dinámicas de TADs

```
#include "Punto.h"
#include "Circulo.h"
int main(){
    Punto p1;
    Circulo c(p1);
    return 0;
}
```



¿Qué métodos se ejecutan?

Implementaciones estáticas y dinámicas de TADs

```
Punto(float xx, float yy){  
    _x = new float;      *_x = xx;  
    _y = new float;      *_y = yy;  
}
```

```
int main(){  
    Punto p1;           Circulo c(p1);  
    return 0;  
}
```



¿Qué ocurre si cambiamos el constructor de Punto?

Implementaciones estáticas y dinámicas de TADs

```
Circulo(const Punto& p, float r=0) {
    _radio = new float;          *_radio = r;
    _centro = new Punto;        *_centro = p;
}
int main(){
    Punto p1;          Circulo c(p1);
    return 0;
}
```



¿Qué ocurre si lo que cambiamos es el parámetro Punto del constructor de Circulo a una referencia constante?

Implementaciones estáticas y dinámicas de TADs

- Implementación dinámica del TAD Conjunto (array dinámico)

```
class Conjunto {
    static const int MAX = 50;
    int _tam;
    int *_elementos;
public:
    // Constructor
    Conjunto();
    // Destructor
    ~Conjunto();
    // Constructor de copia
    Conjunto (const Conjunto& m);
    // Operador de asignación
    Conjunto& operator= (const Conjunto& m);
```

Conjunto.h

Implementaciones estáticas y dinámicas de TADs

```
// inserta un elemento (mutadora)
bool inclusion(int e);
// elimina un elemento (mutadora)
bool exclusion(int e);
// si el elemento pertenece al conjunto, devuelve 'true'
// (observadora)
bool pertenencia(int e) const;
// indica si el conjunto está vacío (observadora)
bool esVacio() const;
// otras operaciones (p.e., operaciones de combinación de
// conjuntos: unión, intersección, diferencia)
};
```

Implementaciones estáticas y dinámicas de TADs

```
Conjunto::Conjunto() { _tam = 0; _elementos = new int[MAX];}
Conjunto::~~Conjunto(){delete[] _elementos;}
Conjunto::Conjunto (const Conjunto& c){
    _tam = c._tam;
    _elementos = new int[MAX];
    for (int i=0; i < _tam; i++)
        _elementos[i] = c._elementos[i];
}
Conjunto& Conjunto::operator= (const Conjunto& c){
    _tam = c._tam;
    for (int i=0; i < _tam; i++)
        _elementos[i] = c._elementos[i];
    return *this;
}
```

Conjunto.cpp

// la implementación se mantiene para el resto de operaciones



TADs genéricos

- TAD genérico = uno o más de los tipos que se usan en la implementación del TAD se dejan sin especificar
 - Se pueden usar las mismas operaciones y estructuras con distintos tipos concretos
- C++ soporta el desarrollo de código genérico mediante el mecanismo de plantillas (**template**)
 - Plantillas de clase: parametriza la definición de un tipo

```
template<class Tipo1, ..., class TipoK>
class nombre_clase {
    //... cuerpo de la clase
}
```

donde **Tipo i** ($i \in [1, K]$)
 - representa el nombre de un tipo admitido como parámetro (actúa como *comodín*) y recibe el nombre de “parámetro tipo” de la plantilla
 - representa a cualquier tipo predefinido o definido por el usuario
 - puede ser usado en el ámbito de la clase

- Instanciación de una plantilla de clase
 - La plantilla de clase no hace que el compilador genere código
 - Instanciación = Proceso que se resuelve en tiempo de compilación y provoca la creación de una versión específica de la plantilla, es decir, una clase concreta (llamada *clase plantilla*)
 - La instanciación de plantillas de clase es explícita
 - Una clase concreta se representa mediante el nombre de la clase genérica seguida de los argumentos de plantilla entre < >

TADs genéricos

- TAD Conjunto genérico

```
#ifndef _CONJUNTO_GENERICO_H_
#define _CONJUNTO_GENERICO_H_
template<class T>
class Conjunto {
    static const int MAX = 100;
    int _tam;
    T _elementos[MAX];
public:
    Conjunto();
    bool inclusion(const T& e); // mutadora parcial
    bool exclusion(const T& e); // mutadora parcial
    bool pertenencia(const T& e) const; // observadora
    bool esVacio() const; // observadora
    // otras operaciones...
};
```

Conjunto_generico.h



Por tema de eficiencia, lo que antes eran parámetros por valor se han cambiado a parámetros por referencia constante

TADs genéricos

```
template<class T>
Conjunto<T>::Conjunto() { _tam = 0; }
```

Conjunto_generico.h

```
template<class T>
bool Conjunto<T>::inclusion(const T& e) {
    bool error = false;
    if (_tam == MAX) error = true;
    else {
        _elementos[_tam] = e;
        _tam ++;
    }
    return error;
}
```

En el caso de los TADs genéricos:

- la implementación de los métodos debe realizarse en el .h: en caso contrario se producirán errores de enlazado
- se pueden implementar en la propia definición de la clase o fuera de la clase, como se hace aquí
- si la implementación del método va fuera de la clase debe ir precedida de la especificación de plantilla y el nombre del método debe ir cualificado con el nombre genérico de la clase (el nombre seguido del tipo genérico entre ángulos)

TADs genéricos

```
template<class T>
bool Conjunto<T>::exclusion(const T& e) {
    bool esta = false;
    int i=0;
    while (i<_tam)
        if (_elementos[i]==e){
            _elementos[i] = _elementos[_tam-1];
            _tam --;
            esta = true;
        }
        else i++;
    return !esta;
}
```

Conjunto_generico.h

TADs genéricos

```
template<class T>
bool Conjunto<T>::pertenencia(const T& e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta){
        esta = _elementos[i]==e;
        i++;
    };
    return esta;
}
```

Conjunto_generico.h

```
template<class T>
bool Conjunto<T>::esVacio() const{
    return _tam == 0;
}
#endif
```



TADs genéricos

```
#include <iostream>
using namespace std;
#include "conjunto_generico.h"

int main() {

    Conjunto<int> c;

    if (c.esVacio()) cout << "El conjunto esta vacio";
    // resto de código
    return 0;
}
```



Operaciones parciales: Tratamiento de errores

- Distintas estrategias de tratamiento de errores
 - Mostrar un mensaje por pantalla y devolver cualquier valor
 - Fácil de implementar ¡pero poco elegante!
 - Admisible sólo cuando estamos depurando nuestro código, la librería es muy pequeña y somos su único usuario
 - Devolución de valor de error (manejo de errores *a lo FP*)
 - Requiere que haya un valor de error disponible que no pueda ser confundido con una respuesta de no-error (típicamente NULL , -1, etc.) o devolver un booleano indicando el estado de error
 - El código cliente debe verificar mediante algún condicional si ha habido o no error
 - Lanzar una excepción
 - Requiere soporte del lenguaje de programación (los lenguajes con OO suelen soportar excepciones)
 - El código cliente debe decidir si quiere manejar la excepción lanzada
 - El código cliente que maneja excepciones es más limpio que en el caso anterior

Operaciones parciales: Tratamiento de errores

- Ejemplo de manejo de excepciones: TAD Conjunto genérico y excepciones de tipo cadena de caracteres

```
#ifndef _CONJUNTO_GENERICO_H_
#define _CONJUNTO_GENERICO_H_
template<class T>
class Conjunto {
    static const int MAX = 50;
    int _tam;
    T _elementos[MAX];
public:
    Conjunto();
    void inclusion(const T& e);
    void exclusion(const T& e);
    bool pertenencia(const T& e) const;
    bool esVacio() const;
    // otras operaciones...
};
```

Conjunto_generico.h



Las operaciones parciales inclusion y exclusion ahora no devuelven el error, lanzarán una excepción

Operaciones parciales: Tratamiento de errores

```
template<class T>
Conjunto<T>::Conjunto() { _tam = 0; }
```

Conjunto_generico.h

```
template<class T>
void Conjunto<T>::inclusion(const T& e) {
    if (_tam == MAX) throw "Conjunto lleno";
    _elementos[_tam] = e;
    _tam ++;
}
```



Tipo de las excepciones lanzadas:
cadena de caracteres

Operaciones parciales: Tratamiento de errores

```
template<class T>
void Conjunto<T>::exclusion(const T& e) {
    bool esta = false;
    int i=0;
    while (i<_tam)
        if (_elementos[i]==e){
            _elementos[i] = _elementos[_tam-1];
            _tam --;
            esta = true;
        }
        else i++;
    if (!esta) throw "Elemento inexistente";
}
```

Conjunto_generico.h

Operaciones parciales: Tratamiento de errores

```
template<class T>
bool Conjunto<T>::pertenencia(const T& e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta){
        esta = _elementos[i]==e;
        i++;
    };
    return esta;
}
```

Conjunto_generico.h

```
template<class T>
bool Conjunto<T>::esVacio() const{
    return _tam == 0;
}
#endif
```

Operaciones parciales: Tratamiento de errores

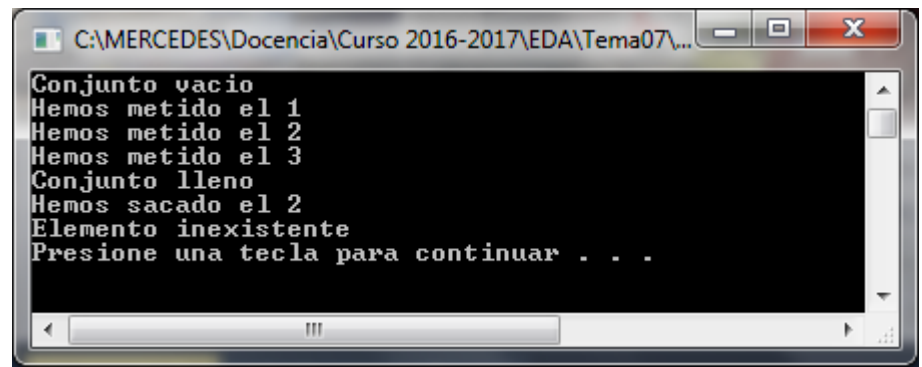
```
int main() {
    Conjunto<int> c;
    if (c.esVacio()) cout << "Conjunto vacio" << endl;
    try{
        c.inclusion(1);
        cout << "Hemos metido el 1\n";
        c.inclusion(2);
        cout << "Hemos metido el 2\n";
        c.inclusion(3);
        cout << "Hemos metido el 3\n";
        c.inclusion(4);
        cout << "Hemos metido el 4??\n";
    }
    catch (const char* &cadena) {
        cout << cadena << endl;
    }
}
```

Para este ejemplo se ha cambiado MAX a 3

Operaciones parciales: Tratamiento de errores

```
try{
    c.exclusion(2);
    cout << "Hemos sacado el 2\n";
    c.exclusion(2);
    cout << "Hemos sacado el 2... de nuevo??\n";
}
catch (const char* &cadena) {
    cout << cadena << endl;
}

return 0;
}
```



```
C:\MERCEDES\Docencia\Curso 2016-2017\EDA\Tema07\...
Conjunto vacio
Hemos metido el 1
Hemos metido el 2
Hemos metido el 3
Conjunto lleno
Hemos sacado el 2
Elemento inexistente
Presione una tecla para continuar . . .
```

Operaciones parciales: Tratamiento de errores

- Ejemplo de manejo de excepciones: TAD Conjunto genérico y clases de excepciones

```
#ifndef _CONJUNTO_GENERICO_H_
#define _CONJUNTO_GENERICO_H_
// Excepción de conjunto lleno
class ConjuntoLleno {};
// Excepción de elemento inexistente
class ElementoInvalido {};
```

```
template<class T>
class Conjunto {
    static const int MAX = 50;
    int _tam;
    T _elementos[MAX];
public:
    Conjunto();
    void inclusion(const T& e);
    void exclusion(const T& e);
    bool pertenencia(const T& e) const;
    bool esVacio() const;
    // otras operaciones...
};
```

Conjunto_generico.h

Uso de clases específicas en las excepciones lanzadas: aumenta la legibilidad y facilita el suministro de información dentro de las propias excepciones

Operaciones parciales: Tratamiento de errores

```
template<class T>
Conjunto<T>::Conjunto() { _tam = 0; }
```

Conjunto_generico.h

```
template<class T>
void Conjunto<T>::inclusion(const T& e) {
    if (_tam == MAX) throw ConjuntoLleno();
    _elementos[_tam] = e;
    _tam ++;
}
```

Operaciones parciales: Tratamiento de errores

```
template<class T>
void Conjunto<T>::exclusion(const T& e) {
    bool esta = false;
    int i=0;
    while (i<_tam)
        if (_elementos[i]==e){
            _elementos[i] = _elementos[_tam-1];
            _tam --;
            esta = true;
        }
        else i++;
    if (!esta) throw ElementoInvalido();
}
```

Conjunto_generico.h

Operaciones parciales: Tratamiento de errores

```
template<class T>
bool Conjunto<T>::pertenencia(const T& e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta){
        esta = _elementos[i]==e;
        i++;
    };
    return esta;
}
```

Conjunto_generico.h

```
template<class T>
bool Conjunto<T>::esVacio() const{
    return _tam == 0;
}
#endif
```

Operaciones parciales: Tratamiento de errores

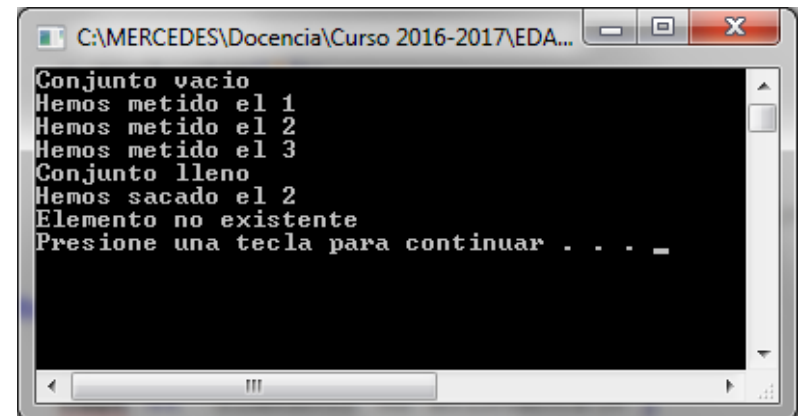
```
int main() {
    Conjunto<int> c;
    if (c.esVacio()) std::cout << "Conjunto vacio\n";
    try{
        c.inclusion(1);
        std::cout << "Hemos metido el 1\n";
        c.inclusion(2);
        std::cout << "Hemos metido el 2\n";
        c.inclusion(3);
        std::cout << "Hemos metido el 3\n";
        c.inclusion(4);
        std::cout << "Hemos metido el 4??\n";
    }
    catch (ConjuntoLleno) {
        //... código de manejo del error
        std::cout << "Conjunto lleno\n";
    }
}
```

Para este ejemplo se ha cambiado MAX a 3

Operaciones parciales: Tratamiento de errores

```
try{
    c.exclusion(2);
    std::cout << "Hemos sacado el 2\n";
    c.exclusion(2);
    std::cout << "Hemos sacado el 2... de nuevo??\n";
}
catch (ElementoInvalido) {
    //... código de manejo del error
    std::cout << "Elemento no existente\n";
}

system("PAUSE");
return 0;
}
```



```
C:\MERCEDDES\Docencia\Curso 2016-2017\EDA...
Conjunto vacio
Hemos metido el 1
Hemos metido el 2
Hemos metido el 3
Conjunto lleno
Hemos sacado el 2
Elemento no existente
Presione una tecla para continuar . . . _
```



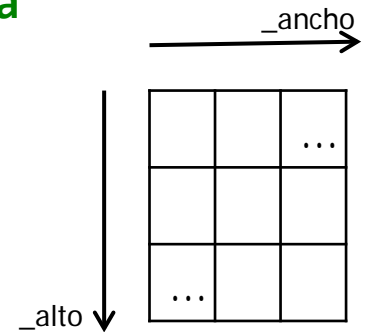
Operaciones parciales: Tratamiento de errores

- Excepciones en constructores
 - Las excepciones proporcionan una solución al problema de devolver un error cuando algo falla en un constructor
 - Recuerda... los constructores no devuelven nada
 - Ten en cuenta que un objeto no se considera construido hasta que su constructor se ha completado
 - Entonces y sólo entonces al desapilar se llamará al destructor del objeto
 - El destructor no se invoca cuando un objeto no se ha construido completamente

Operaciones parciales: Tratamiento de errores

Ejemplo: TAD Mapa y excepciones en constructor (clase propia)

```
class Mapa {
    unsigned short *_celdas; // 2 bytes por celda
    int _ancho; // numero de columnas
    int _alto; // numero de filas
public:
    class DimensionesInvalidas {};
    // Constructor
    Mapa(int ancho, int alto): _ancho(ancho), _alto(alto){
        if (ancho < 1 || alto < 1)
            throw DimensionesInvalidas();
        _celdas = new unsigned short[_alto*_ancho];
        for (int i=0; i<_alto*_ancho; i++) _celdas[i] = 0;
        std::cout << "Construido mapa de dimensiones "
                  << _ancho << "x" << _alto << "\n";
    }
}
```



Operaciones parciales: Tratamiento de errores

```
// Destructor
~Mapa() {
    std::cout << "Destruyendo mapa de dimensiones "
                << _ancho << "x" << _alto << "\n";
    delete[] _celdas;
}

// resto de métodos...
};
```

Operaciones parciales: Tratamiento de errores

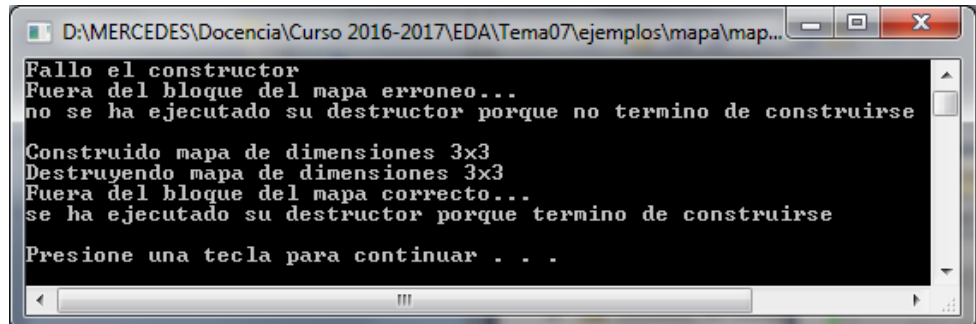
```
int main(){

    // bloque para detección de fallo en constructor
    try{

        Mapa mapa_erroneo(0,0);
    }catch (Mapa::DimensionesInvalidas){
        std::cout << "Fallo el constructor\n";
    }

}

std::cout << "Fuera del bloque del mapa erroneo...\n";
std::cout << "no se ha ejecutado su destructor porque no
termino de construirse\n\n";
```



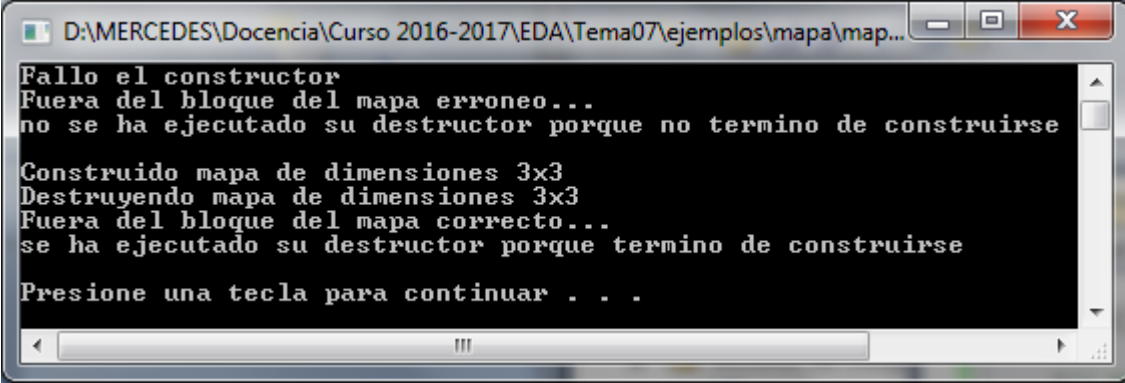
```
D:\MERCEDES\Docencia\Curso 2016-2017\EDA\Tema07\ejemplos\mapa\map...
Fallo el constructor
Fuera del bloque del mapa erroneo...
no se ha ejecutado su destructor porque no termino de construirse

Construido mapa de dimensiones 3x3
Destruyendo mapa de dimensiones 3x3
Fuera del bloque del mapa correcto...
se ha ejecutado su destructor porque termino de construirse

Presione una tecla para continuar . . .
```

Operaciones parciales: Tratamiento de errores

```
{// bloque para detección de fallo en constructor
  try{
      Mapa mapa_correcto(3,3);
  }catch (Mapa::DimensionesInvalidas){
      std::cout << "Fallo el constructor";
  }
}
std::cout << "Fuera del bloque del mapa correcto...\n";
std::cout << "se ha ejecutado su destructor porque
termino de construirse\n\n";
return 0;
}
```



```
D:\MERCEDES\Docencia\Curso 2016-2017\EDA\Tema07\ejemplos\mapa\mapa...
Fallo el constructor
Fuera del bloque del mapa erroneo...
no se ha ejecutado su destructor porque no termino de construirse

Construido mapa de dimensiones 3x3
Destruyendo mapa de dimensiones 3x3
Fuera del bloque del mapa correcto...
se ha ejecutado su destructor porque termino de construirse

Presione una tecla para continuar . . .
```

Operaciones parciales: Tratamiento de errores

Ejemplo: TAD Mapa y uso de excepciones estándar en constructor

```
class Mapa {
    unsigned short *_celdas;
    int _ancho;
    int _alto;
public:
    // Constructor
    Mapa(int ancho, int alto): _ancho(ancho), _alto(alto){
        if (ancho < 1 || alto < 1)
            throw std::invalid_argument("Dimensiones erroneas");
        _celdas = new unsigned short[_alto*_ancho];
        for (int i=0; i<_alto*_ancho; i++)
            _celdas[i] = 0;
        std::cout << "Construido mapa de dimensiones "
                    << _ancho << "x" << _alto << "\n";
    }
}
```

Operaciones parciales: Tratamiento de errores

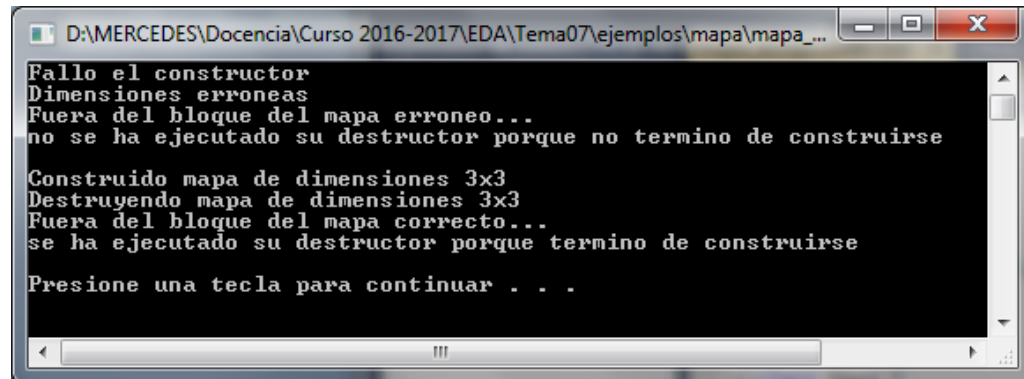
```
// Destructor
~Mapa() {
    std::cout << "Destruyendo mapa de dimensiones "
                << _ancho << "x" << _alto << "\n";
    delete[] _celdas;
}

// resto de métodos...
};
```


Operaciones parciales: Tratamiento de errores

```
int main(){
    // detección de fallo en constructor
    try{
        Mapa mapa_erroneo(0,0);
    }catch (const std::invalid_argument &ia){
        std::cout << "Fallo el constructor\n";
        std::cout << ia.what() << "\n";
    }
}

std::cout << "Fuera del bloque del mapa erroneo...\n";
std::cout << "no se ha ejecutado su destructor porque no termino
de construirse\n\n";
```



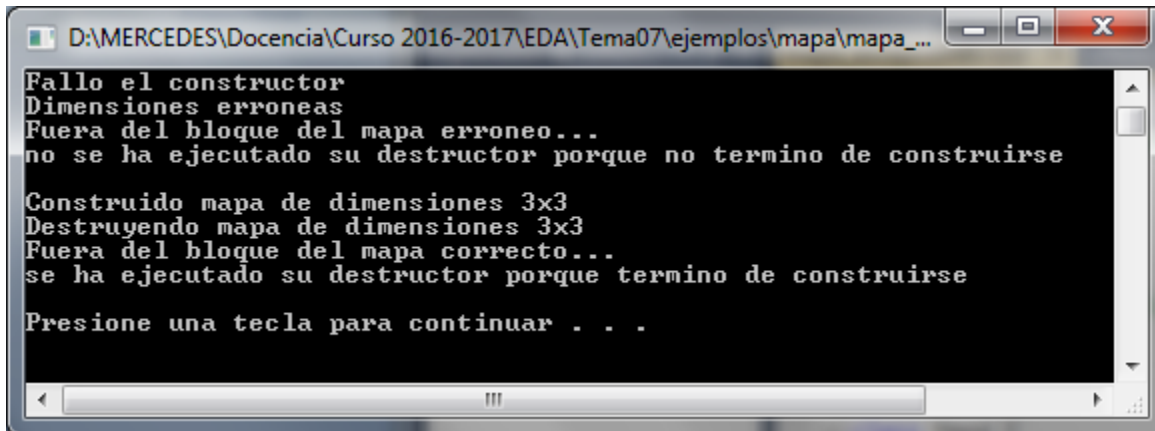
```
D:\MERCEDDES\Docencia\Curso 2016-2017\EDA\Tema07\ejemplos\mapa\mapa_...
Fallo el constructor
Dimensiones erroneas
Fuera del bloque del mapa erroneo...
no se ha ejecutado su destructor porque no termino de construirse

Construido mapa de dimensiones 3x3
Destruyendo mapa de dimensiones 3x3
Fuera del bloque del mapa correcto...
se ha ejecutado su destructor porque termino de construirse

Presione una tecla para continuar . . .
```

Operaciones parciales: Tratamiento de errores

```
{// detección de fallo en constructor
    try{
        Mapa mapa_correcto(3,3);
    }catch (const std::invalid_argument &ia){
        std::cout << "Fallo el constructor";
        std::cout << ia.what() <<"\n";
    }
}
std::cout << "Fuera del bloque del mapa correcto...\n";
std::cout << "se ha ejecutado su destructor porque termino de
construirse\n\n";
return 0;
}
```



```
D:\MERCEDES\Docencia\Curso 2016-2017\EDA\Tema07\ejemplos\mapa\mapa_...
Fallo el constructor
Dimensiones erroneas
Fuera del bloque del mapa erroneo...
no se ha ejecutado su destructor porque no termino de construirse

Construido mapa de dimensiones 3x3
Destruyendo mapa de dimensiones 3x3
Fuera del bloque del mapa correcto...
se ha ejecutado su destructor porque termino de construirse

Presione una tecla para continuar . . .
```

- Pruebas formales
 - Verifican la corrección teórica del software (algoritmos/TADs), desde el punto de vista de su especificación, sin necesidad de acceso a una implementación concreta
 - Algoritmos: Especificaciones pre-post y verificación
 - TADs: Especificaciones algebraicas
 - ¡Muy trabajosas!
 - En escenarios reales sólo se usa en campos de aplicación donde la seguridad e integridad del software son críticas (ej., aviónica, control de centrales nucleares)

- Pruebas de caja negra
 - Validan una implementación basándose en la especificación, es decir, desde el punto de vista de la interfaz del software
 - ¿Funciona todo bien “desde fuera de la caja”?
 - Se suministran datos como entrada y se estudia la salida, sin preocuparse de lo que pueda estar haciendo el software “por dentro”
 - Se contrastan comportamientos esperados con comportamientos obtenidos
 - Idealmente, para cada aspecto debería comprobarse que se produce la salida esperada para cada tipo de entrada con el que se puede enfrentar
 - En la práctica es imposible probar todos los casos y hay que elegir un subconjunto representativo

- Pruebas de caja blanca
 - Validan aspectos concretos de la implementación, teniendo en cuenta el código de la misma
 - Partiendo del conocimiento del código, ejercitan un conjunto básico de caminos o trazas de ejecución para comprobar que funcionan como se espera
 - Camino de ejecución: secuencia concreta de instrucciones que se ejecutan para una entrada concreta



Documentando TADs

- Documentación de un TAD
 - Descripción de alto nivel de la abstracción usada
 - Descripción de cada elemento público
- La documentación va en el .h
- Se sugiere el uso de anotaciones tipo Doxygen (www.doxygen.org/)
 - El sistema Doxygen permite generar documentación pdf y html detallada a partir de los fuentes y de las anotaciones que contienen



Documentando TADs

```
/**
 * @file conjunto.h
 * @author Mercedes Gomez
 * @date 2016-12-01
 * @brief conjunto sencillo
 *
 * Conjunto implementado sobre array
 */
#ifndef _CONJUNTO_GENERICO_H_
#define _CONJUNTO_GENERICO_H_
/// Excepcion de conjunto lleno
class ConjuntoLleno {};
/// Excepcion de elemento inexistente
class ElementoInvalido {};
```

```
template<class T>
class Conjunto {
    static const int MAX = 3; /** < maximo numero de elementos*/
    int _tam; /** < numero de elementos actuales*/
    T _elementos[MAX]; /** < vector estatico de elementos*/

public:
    /**
     * inicializa el conjunto a vacio
     * constructor
     */
    Conjunto();
};
```



```
/**
 * inserta un elemento
 * mutadora parcial: si se llena, lanza
 ConjuntoLleno
 * @param e elemento a insertar
 */
void inclusion(const T& e);
/**
 * elimina un elemento
 * mutadora parcial: si no existe el elemento,
 lanza ElementoInvalido
 * @param e elemento a eliminar
 */
void exclusion(const T& e);
```

```
/**
 * devuelve true si el elemento pertenece al
 conjunto
 * observadora
 * @param e elemento a comprobar
 */
bool pertenencia(const T& e) const;
/**
 * devuelve true si el conjunto esta vacio
 * observadora
 */
bool esVacio() const;
};
```

- Implementa un programa que diga si un número N es feliz o infeliz

Dado N , se cogen sus dígitos y se suman sus cuadrados, dando un número N_1 . El proceso se repite con N_1 y con los siguientes N_i obtenidos mediante ese mecanismo, hasta que:

- Se obtiene un N_i igual a 1, y se dice que el número es feliz
- Nunca se obtiene un 1 (se entra en un ciclo en el que se repite la obtención de una sucesión de números que no incluye al 1), y se dice entonces que el número es infeliz

Por ejemplo:

- El 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
- El 38 es infeliz: $38 \rightarrow 73 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58 \rightarrow \dots$

Uso de TADs

```
#include "conjunto_generico.h"
#include <iostream>
using namespace std;

int cuadrados_digitos(int n) {
    int suma = 0;
    while (n > 0) {
        int digito = n % 10;
        suma = suma + digito*digito;
        n = n / 10;
    }
    return suma;
}
```

```
void psicoanaliza(int n) {
    Conjunto<int> c;
    while (!c.pertenencia(n)) {
        c.inclusion(n);
        n = cuadrados_digitos(n);
    }
    if (n==1) cout << "feliz (llega a 1) " << endl;
    else
        cout << "infeliz (repite del " << n
            << " en adelante)" << endl;
}

int main() {
    psicoanaliza(7);  psicoanaliza(38);
    return 0;
}
```



Bibliografía del tema

- Apuntes ISBN 978-84-697-0852-1
- Object-oriented programming in C++, fourth edition / Robert Lafore / SAMS, 2005
- Cómo programar en C++ / Paul Deitel, Harvey Deitel / Pearson-Prentice Hall, 2014
- C++ : The complete reference / Herbert Schildt / McGraw-Hill, 2003
- Estructuras de datos y métodos algorítmicos: Ejercicios resueltos / Martí Oliet, Ortega Mallén y Verdejo López / Pearson – Prentice Hall, 2010





Acerca de Creative Commons


Licencia CC ([Creative Commons](http://creativecommons.org/))

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:

-  Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.

-  No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.

-  Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

En <http://es.creativecommons.org/> y <http://creativecommons.org/> puedes saber más de Creative Commons.