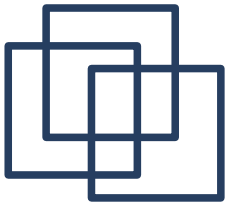
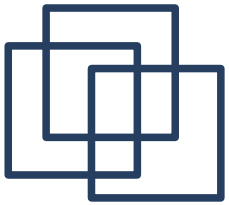
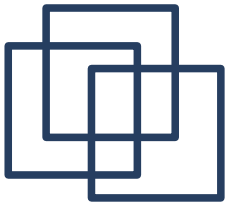


Abstracción de Datos



Tema 2. Abstracción

- Introducción
- Abstracción en programación
- Abstracción de datos: Tipo Dato Abstracto
- TDA en C++
- Abstracción por generalización
- Abstracciones de iteración:
  Contenedores e iteradores

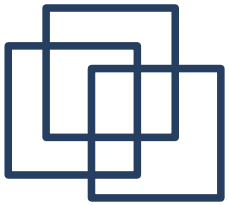


Introducción

Abstraer: *(diccionario RAE) (Del lat. Abstrahĕre). Separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción*

Abstracción en la resolución de problemas:

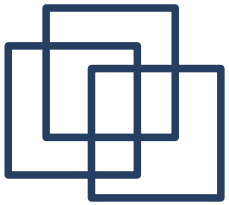
- Ignorar detalles específicos buscando generalidades para favorecer su resolución.
- Descomposición en que se varía el nivel de detalle.



Introducción

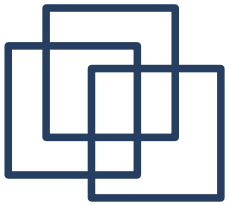
Propiedades de una descomposición útil:

- 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.



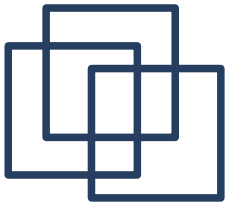
Abstracción en Programación

- Abstracción Procedimental.
  - Conjunto de operaciones (procedimiento) que se comporta como una operación.
  - *Permite considerar (y utilizar) un conjunto de operaciones de cálculo como una operación simple.*
    - En lenguaje de programación: sqrt(double x);
    - Propias: ordenar(int *v, int n);
- La identidad de los datos no es relevante para el diseño. Sólo interesa el número de parámetros y su tipo
- Abstracción por especificación: es irrelevante la implementación, pero NO qué hace



• *Características A.P.*

- ***Localidad*: Para implementar una abstracción procedimental no es necesario conocer la implementación de otras que se use, sólo su especificación.**
- ***Modificabilidad***: Se puede cambiar la implementación de una abstracción procedimental sin afectar a otras abstracciones que la usen, siempre y cuando no cambie la especificación.



Especificación de una A.P.

- **Cabecera**: (Parte sintáctica)

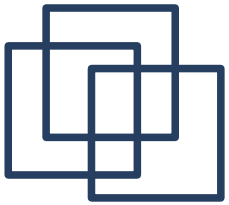
Indica el nombre el procedimiento y el número, orden y tipo de las entradas y salidas. Se suele adoptar la sintaxis de un lenguaje de programación concreto.

- **Cuerpo**: (Parte semántica)

Compuesto por:

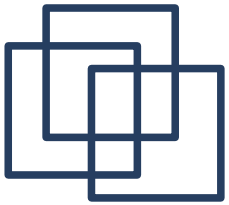
  - Argumentos
  - Requiere
  - Valores de retorno
  - Efecto

- Usar herramientas como DOXYGEN



Ejemplo

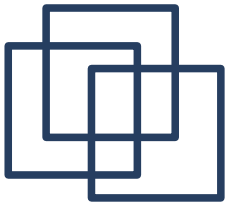
```
/**

@brief Copia bytes desde un area origen de memoria a otra destino

*/

size memcpy(void *dest, const void *src, size n);
```



Especificación A.P.:Argumentos

- Explica el significado de cada parámetro de la abstracción; indica qué representa.
- Expresa las restricciones sobre los datos sobre los que puede operar. Procedimiento *total* y *parcial*.
- Indica si cada argumento es modificado o no.

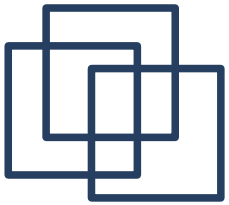
```
/**
@brief Copia bytes desde un area origen de memoria a otra destino

@param[out] dest El area de memoria destino.

@param[in]   src   El area de memoria origen.

@param[in]   n      Numero de bytes a copiar

*/

size memcpy(void *dest, const void *src, size_t n);
```



Especificación A.P.: Requiere

- Restricciones impuestas por el uso del procedimiento no recogidas por *Argumentos*. Ej.: que previamente se haya ejecutado otra abstracción procedimental.

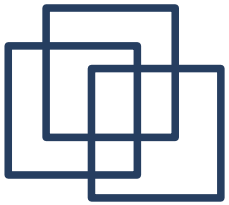
```
/**
@brief Copia bytes desde un area origen de memoria a otra destino

@param[out] dest El area de memoria destino.

@param[in]   src   El area de memoria origen.

@param[in]   n      Numero de bytes a copiar

@pre Las areas de memoria no pueden estar solapadas

*/

size memcpy(void *dest, const void *src, size n);
```



Especificación A.P.: Valores de retorno

- Descripción de qué valores devuelve la abstracción y qué significan.

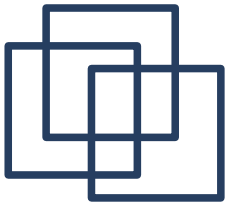
```
/**
@brief Copia bytes desde un area origen de memoria a otra destino
@param[out] dest El area de memoria destino.
@param[in]  src  El area de memoria origen.
@param[in]  n    Numero de bytes a copiar
@pre Las areas de memoria no pueden estar solapadas
@return El numero de bytes que ha copiado
*/
size memcpy(void *dest, const void *src, size n);
```



Especificación A.P.: Efecto

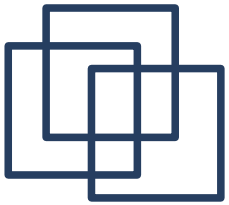
- Describe el comportamiento del procedimiento para las entradas no excluidas por los requisitos. El efecto debe indicar las salidas producidas y las modificaciones producidas sobre los argumentos marcados con “Es MODIFICADO”.

```
/**
@brief Copia bytes desde un area origen de memoria a otra destino

@param[out] ....

@return El numero de bytes que ha copiado

@post Todo el bloque de memoria entre dest y dest+n es modificado

*/

size memcpy(void *dest, const void *src, size n);
```



Especificación junto al código fuente

- Falta de herramientas para mantener y soportar el uso de especificaciones → Responsabilidad exclusiva del programador
- Necesidad de vincular código y especificación
- Incluirla entre comentarios en la parte de interfaz del código



Datos Abstractos (TDA)

TDA: Entidad abstracta formada por un conjunto de datos y una colección de operaciones asociadas.

TDA = (objetos, operaciones)

- Permite extender el lenguaje con nuevos tipos

Ej.: alumno, termino, ...

Nos abstraemos de la representación utilizada. El código que los usa depende sólo de la especificación del tipo (y las operaciones).

USUARIO

El usuario necesita conocer cómo utilizar la interfaz

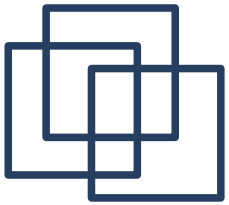
INTERFACE
- volumen (+/-)
- programa (+/-)
- ajuste imagen
- entradas (AV)

IMPLEMENTACION
- monitor de plasma
- Pioneer
- altavoces 40w

La clase necesita conocer qué interfaz implementar

CLASE

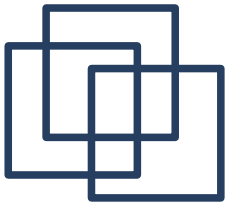
Se puede modificar la **INTERFAZ** sin modificar la interfaz



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.



Abstracción en Programación

Metodología de Diseño: Estrategia Top-Down

- Descomponer el problema en subproblemas más simples, identificando cómo se relacionan los subproblemas.

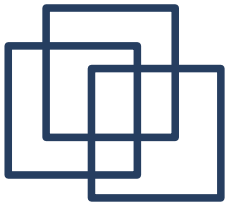
a) Identificar los elementos de información

b) Para cada elemento identificar:

   a) Tipos:

      - Identificar tipos conocidos (nivel conceptual)
      - Para cada tipo desconocido, especificar su dominio y las operaciones que sean relevantes

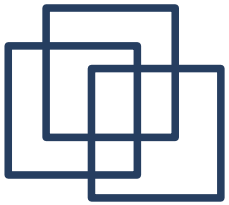
   b) Operaciones (Abstr. Procedimental)



Es útil distinguir entre:

- *Creadores de clases* (aquellos que crean nuevos tipos de datos): Su objetivo es construir una clase que exponga sólo lo necesario para el cliente y mantenga todo lo demás oculto.
- *Programadores clientes* (los consumidores que usan los tipos de datos). Su objetivo es tener un conjunto de herramientas para un rápido desarrollo de aplicaciones.

- ¿Por qué?
  - Porque si está oculto, el creador de clases puede cambiar la parte oculta sin preocuparse de las consecuencias sobre los demás.

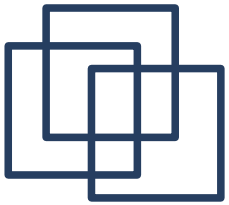


Datos Abstractos (TDA)

TDA = (objetos, operaciones)

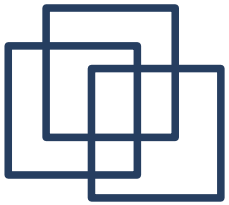
Conceptos manejados:

- **Especificación**: Describe el 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

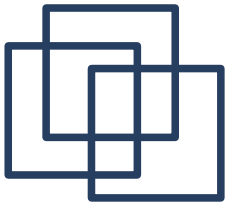
- Visión externa: especificación
- Visión interna: representación e implementación
- Ventajas de la separación:
  - Se puede cambiar la visión interna sin afectar a la externa
  - Facilita la labor del programador permitiéndole concentrarse en cada fase por separado



Especificación de un TDA

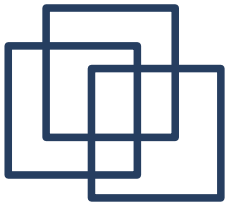
La especificación es esencial. Define su comportamiento, pero no dice nada sobre su implementación.

Indica el tipo de entidades que modela, qué operaciones se les pueden aplicar, cómo se usan y qué hacen.



Especificación de un TDA

- Estructura de la especificación:
  - **Cabecera**: nombre del tipo y listado de las operaciones.
  - **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.
  - **Operaciones**: Especificar las operaciones una por una como abstracciones procedimentales.



Tipo abstracto: Tipos de operaciones

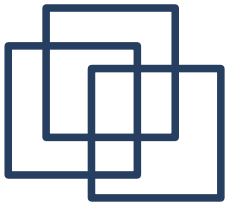
1) **Constructores primitivos**. Crean objetos del tipo sin necesitar objetos del mismo tipo como entrada.

2) **Constructores de copia**. Crean objetos del tipo a partir de otros objetos del tipo.

3) **Modificadores o mutadores**. Operadores que modifican los objetos del tipo.

4) **Observadores o consultores**. Toman como entrada objetos de un tipo y devuelven objetos de otro tipo.

- Es usual la combinación de operadores. En particular, de los tipos 3 y 4.



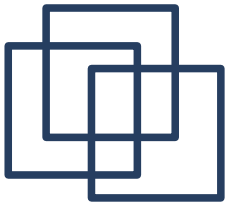
Clase Fecha: Especificación

- Esti tipo de dato nos permite almacenar fechas del calendario, en el formato dd/mm/aaaa.
- Constructor

```
Fecha ( int day, int month, int year);
```

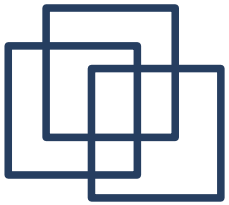
- Consultores

```
// @return  mes almacendo en el objeto
 int getMonth()
// @return: dia almacenado en el objeto
 int getDay()
// @returns: Año almacenado
 int getYear()
```



Clase Fecha: Uso

```
Void ejemplo ( ) {
 Fecha x(06,04,1980);
cout << “El día es “ << x.getDay() << endl;
cout << “El mes es “ << x.getMonth() << endl;
cout << “El año es “ << x.getYear() << endl;
...
Cout << “ Han pasado “ << y.getDay() -X.getDay() <<
“dias” << endl;
}
```

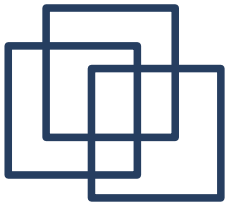


Representación de un TDA

- Forma concreta en que se representan los datos en un lenguaje de programación para poder manipularlos
- En la representación hay dos tipos de datos:
  - Tipo *Abstracto*: definido en la especificación

    Tipo (concepto) fecha

  - Tipo *rep*: tipo a usar para representar los objetos del tipo abstracto y sobre él implementar las operaciones. No es único



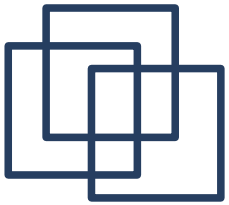
Clase Fecha: Representación

- Representacion I:

```
int month;
int day;
int year;
```

- Representacion II

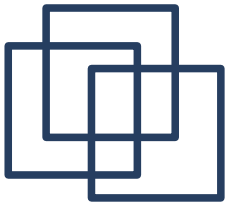
```
Int day;
String month;
Char anio[4];
```



Ayuda para entender la representación

Es importante:

- Conectar ambos tipos (Función de abstracción)
- Identificar los objetos del tipo rep que representan objetos abstractos válidos (Invariante de representación)



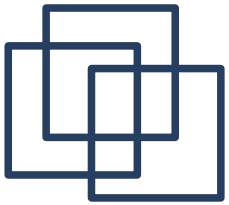
Función de abstracción

- Captura las ideas que han llevado al diseñador del TDA a utilizar la representación concreta.
- Responde a la pregunta:
  - ¿Qué variables se utilizan y como se relacionan con el TDA abstracto?

- Define el significado de un objeto rep de cara a representar un objeto abstracto. La forma en que un objeto del TDA se supone que implementa al objeto abstracto
- Establece una relación formal entre un objeto rep y un objeto abstracto.
- Es una aplicación sobreyectiva:

$$fA: rep \rightarrow TDAbstracto$$

- **Se debe incluir como comentario en la implementación**

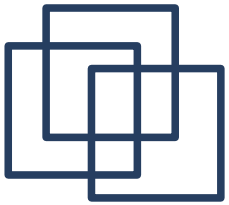


Invariante de Representación

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

IR: rep → bool

- 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:
  - Los objetos creados por constructores lo verifican
  - 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
- **Se debe incluir como comentario en la implementación**



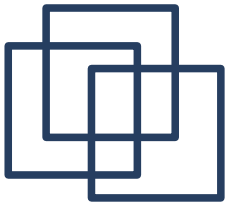
Clase Fecha: Representación

- Representacion

```
int month;

int day;

int year;
```

- AF(d) = d es la fecha concreta day/month//year
- IR(d) = (d.month >= 1) && (d.month <= 12) && (d.year >= 1980) && (d.day es un dia valido para el mes d.month)
- Es bueno incluir un método para verificar que un objeto de tipo fecha satisface el invariante de la representación

```
bool checkRep();
```



Otros ejemplos

Tipo string: (private: char * pc; unsigned int n;)

Funcion de Abstraccion:

AF: Rep => Abs

Una cadena abstracta, cad, se instancia en el string str teniendo:

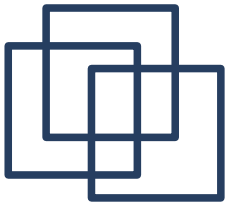
str.n = cad.longitud ;

str.pc[0]==cad[0], …, str.pc[n-1]=cad[cad.longitud-1]

Invariante de la Representacion:

str.n >= 0;

str.pc debe tener un área de memoria reservada de tamaño n



Otros ejemplo

Tipo String:private char *pc:

Funcion de Abstraccion:

AF: Rep => Abs

Una cadena abstracta, cad, se instancia en el string str teniendo:

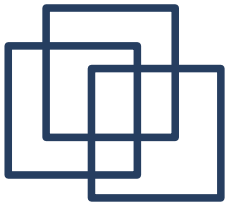
menor i tq str.pc[i]='\0' == cad.length.

str.pc[0]==cad[0], ..., str.pc[n]=cad[cad.length-1]

Invariante de la Representacion:

Para una cadena str, existe un i tq str.pc[i]='\0'

str.pc debe tener un area de memoria reservada de tamaño n+1



Tipo string (char *, char *, char *)

Funcion de Abstraccion:

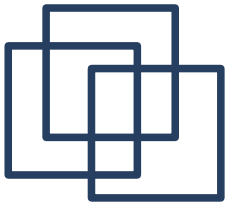
AF: Rep => Abs

(char * inicio, char *final, char * fin_almacenamiento) => cadena

Una cadena abstracta, cad, se instancia en el mi_string str teniendo:

(str.final-str.inicio) = cad.longitud ;

*(str.inicio) ==cad[0], ..., *(str.final)=cad[cad.longitud-1]



Invariante de la Representacion:

(0) inicio <=final <=fin_almacenamieto

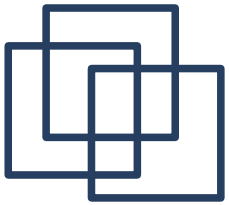
(1) [inicio, final) representan el rango valido de la cadena

(2) *final contiene el caracter '\0'

(3) Cada posicion en [final, fin_almacenamiento) representa memoria reservada, pero no utilizada.

Estas posiciones son utiles para optimizar futuros redimensionamientos.

Como consecuencia, para almacenar una cadena de longitud n necesitamos al menos n+1 posiciones de memoria.



Ejemplos de F.A.

- TDA racional: {num, den} --> num/den
- TDA Fecha:

  {dia, mes, anio} --> dia/mes/anio
- TDA Polinomio:

  r[0..n] --> r[0] + r[1] x + ... + r[n]x^n
- TDA termino:

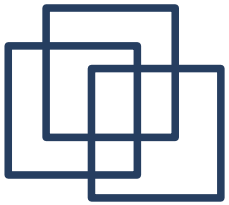
  {int, char *, tama, float} -->

  id: identificador del termino

  char *: cadena que representa el termino

  tama: numero de caracteres del termino

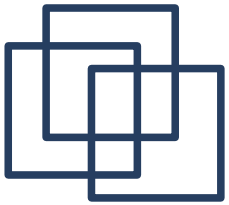
  frec: numero de veces que aparece el termino.



Ejemplos de I.R.

- Dado un objeto racional r: rep r={num,den}

  IR: r.den != 0
- Dado un objeto termino:
  - rep t={id,cadena,tama,frec}
  - debe cumplir:
    - Id >= -1; (-1 indica termino sin id)
    - Cadena[tama] = '\0';
    - Tama >= 0;
    - Frec >= -1; (-1 indica no computada)



Ejemplos de I.R.

- Dado un objeto fecha: rep f={dia,mes, anio} debe cumplir:

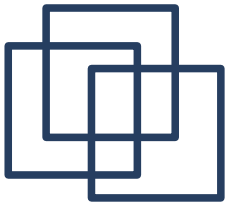
  1 <= dia <= 31

  1 <= mes <= 12

  Si mes == 4,6,9 u 11, entonces dia <= 30

  Si mes == 2 y bisiesto(anio), dia <= 29

  Si mes == 2 y !bisiesto(anio), dia <= 28

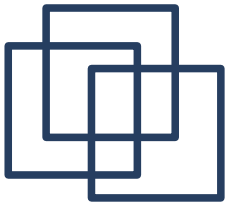


Implementación.

- Utilizando la representación implementar cada uno de los métodos

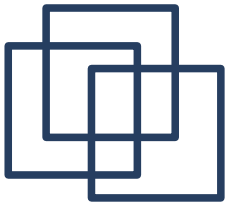
```
Fecha::Fecha (int month, int day, int year) {
  this.month = month;   this.day = day; this.year = year;
   }
 int Fecha::getMonth() {     return month; }
int Fecha::getDay() {    return day; }
int Fecha::getYear() {   return year; }
```

CLASES



Clases en C++ (2)

- ***Ocultamiento de información**: Mecanismo para impedir el acceso a la representación e implementación desde fuera del tipo.*
  - *Garantiza:*
    - Representación inaccesible: razonamientos correctos sobre la implementación.
    - Cambio de representación e implementación sin afectar a la especificación del TDA.
  - Control de acceso a los componentes miembros regulado mediante dos niveles de acceso: público (*public*) y privado (*private*).

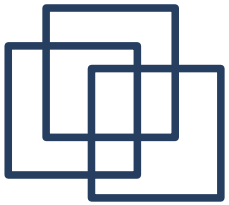


Clases en C++ (3)

- ***Encapsulamiento**: Ocultamiento de información y capacidad para expresar el estrecho vínculo entre datos y operaciones.*

  - Las componentes de una clase son
    - datos (*variables de instancia*)
    - operaciones (*funciones miembro o métodos*).

  La clase establece un fuerte vínculo entre todos los miembros (datos y operaciones). Ambos se indican juntos.

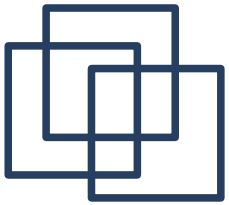


Control de acceso

```
class nombre_clase {
public:
    .....
private:
   ....
};   // Fin de la clase
```

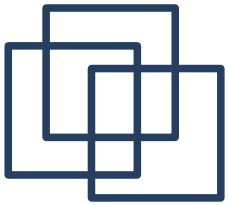
Las clases permiten controlar el acceso a sus componentes (miembros) usando dos nuevas palabras reservadas:

- *private*: indica que estos miembros son accesibles sólo por otros miembros de la clase, pero no por nadie externo a la clase (ocultamiento de información).
- *public*: los miembros definidos son accesibles para cualquiera.



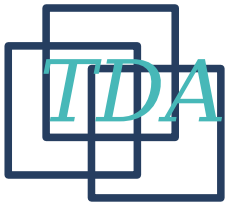
Control de acceso (2)

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



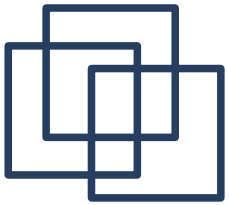
(Clase termino. Versión 0.1)

```
class string{
public:
  string( );
  ~string( );
  unsigned int size( ) const;
  char & at(unsigned int & pos);
  string & append( const char *
  s);
  string & append( const string &
  str);
  ......
private:
 char* cadena;
 int n;
};
```



TDA.h Especificación+Representación

```
#ifndef  TDA_H
#define TDA_H
class tda {
public:
    metodos publicos;
      constructores; (mismo nombre que la clase);
      otros metodos;
   datos publicos;  No recomendable
private:
    datos privados;
    metodos privados;
}
#endif
```



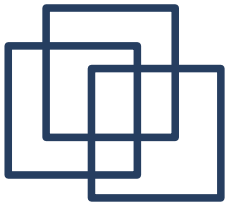
Funciones miembro

- Se declaran dentro de la definición de la clase

```
class tda {
public:
   ...
   void  ponerID(const int  & ID);
  ....
}
```

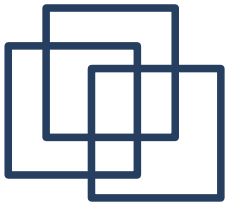
- Se definen fuera. Para indicar que son miembros de su clase se usa el operador de ámbito (::):

```
void tda::ponerID(const int  & ID)
 {...}
```



Un poco de orden

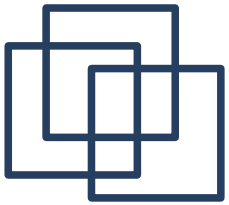
- La definición de las funciones miembro es parte de la implementación.
- Por ello, su definición se incluye en el fichero .cpp y no en el .h.
- El usuario del tipo, no necesita conocer su implementación para usarlas. Su definición sólo es necesaria en tiempo de enlazado, no de compilación.
- Especificación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



Notas sobre el diseño de un TDA

Implica dos tareas:

- a) Diseñar una representación que se va a dar a los objetos.
  - Se elige después de haber hecho la especificación. Existe el riesgo de hacerlo a la inversa.
  - La elección viene condicionada por la sencillez y la eficiencia en las implementaciones de las operaciones del tipo.
- b) Basándose en la representación, implementar cada operación.



Ejemplo: Números Racionales

/** TDA racional

racional::racional, numerador, denominador, =, +=

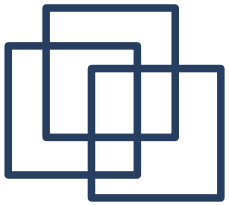
Definición:

Esta clase implementa el TDA numero racional. Un numero racional esta compuesto de un numerador (valor entero) y un denominador (valor entero).

$$R = n/d$$

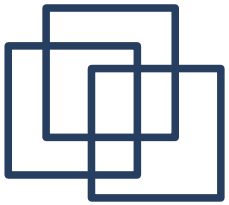
Son objetos mutables, Residen en memoria estática.

*/



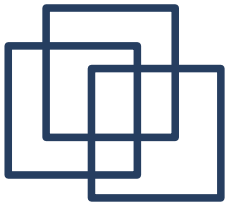
TDA Racional. Operaciones

```
/**   Constructor primitivo.
          @doc Crea un objeto racional con valores 0/1;
*/ racional ();
/* Constructor de inicialización
@param n: entero que representa el numerador
@param d; entero que representa el denominador
@precondition d debe ser distinto de cero
@doc construye el objeto racional n/d
*/ racional (int n, int d);
```



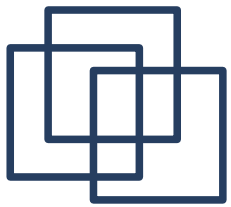
TDA Racional. Operaciones

```
/* Constructor de copia
@param r: numero racional válido
@doc construye  un objeto racional que es copia de r
*/ racional (const racional & r);
/* Operador de asignación
@param r: numero racional válido
@doc asigna el racional r al objeto que llama a la función
*/ void operator=(const racional & r);
.....etc.
```



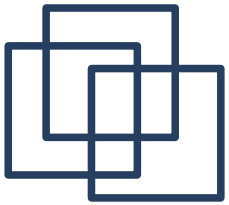
racional.h: Especif+Representación

```
#ifnded __RACIONAL_H
#define __RACIONAL_H
class racional {
public:   //Especificación
     racional();
     racional(int n,int d);
     racional(const racional & r);
     int numerador() const;
     int denominador() const;
     void operator=(const racional &r);
     void operator+=(const racional &r);
```



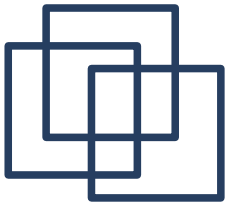
racional.h: Especificación+Representac

```
class racional {

public:

    ........

private:  // Representación

        int num;  // Estos campos solo se pueden acceder a traves de las
        int den;   // funciones miembros de la clase
       void normalizar();
};


#endif
```



TDA racional: Ejemplo de us

```
#include <iostream>
#include “racional.h”
void main() {
    racional r, s(2,5);
    racional t(r);
    cout << s.numerador() << “/” << s.denominador();
    r = s;
    r += t;
    cout << r.numerador() << “/” << t.denominador();
}
```



tda.cpp: Implementación

Este fichero debe comenzar por

#include “tda.h”

una función miembro se identifica como:

- constructores:

  tda::tda([lista de argumentos]);

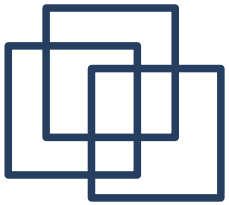
- Metodos genéricos:

  tipo_devuelve tda::nombre_metodo([lista de argumentos]);

- Operadores:

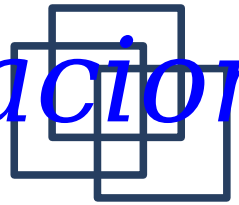
  tipo_devuelve tda::operatorS([lista de argumentos]);

siendo S un simbolo que representa al operador, =, <, <=, ++, +, ==, *, ...



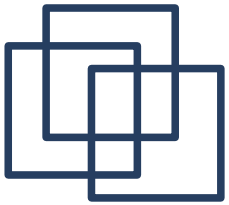
racional.cpp: Implementación

```
#include “racional.h”

racional::racional()
   { num = 0; den = 1;}


racional::racional(int numerador, int
  denominador)
{ num = numerador; den = denominador;
   normalizar(); }


racional::racional(const racional & r)
{ num = r.num; den = r.den ; }
```



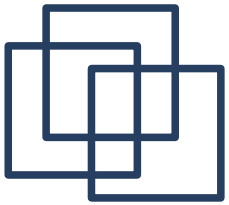
racional.cpp: Implementación

```
int racional::numerador()const
   { return num;}
int racional::denominador() const
   { return den; }
void racional::operator=(const racional & r)
   { num = r.num; den = r.den; }
void racional::operator+=(const racional & r)
   { int arriba, debajo;
     arriba = num*r.den + den*r.num;
     debajo = den * r.den;
     num = arriba; den = debajo;
     normalizar(); }
```



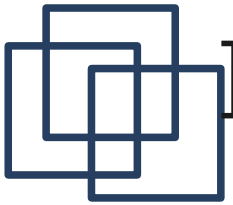
racional.cpp: Implementació

```
void racional::normalizar()
{ int signo,d;
 signo = ((num/den)>0)?1:-1;
 assert (den!=0);
  d = mcd(num,den);
  num = signo * abs(num/d);
  den = abs(den/d);
}
```



Funciones amigas (*friend*)

- En ocasiones es necesario que una función global (no miembro de una clase) tenga acceso a sus miembros privados. Esto implica saltarse los mecanismos de control de acceso.
- C++ ofrece una solución para estas situaciones especiales: *funciones amigas*.
- Se tienen que declarar dentro de la definición de la clase, precediendo su definición con la palabra reservada *friend*.



Funciones amigas (*friend*)

```
class Fecha {
friend ostream & operator<<(ostream &s, const
  Fecha & f);
….
private:
  int dia; int mes; int año;
};
////////////////////////////////////////////////
  ////////////////
en fichero.cpp
ostream & operator<<(ostream &s, const Fecha &
  f){
  s << f.dia<<"/" << f.mes<<"/" << f.año;
 return s;
}
```