

# **Estructuras de Datos y Algoritmos 1**

## **6- Tipos Abstractos de Datos (Introducción)**

# Abstracción

1. Ignorar detalles
2. Posponer la consideración de detalles

## En ciencias experimentales

Se observan fenómenos y se crean modelos abstractos que explican aspectos de esos fenómenos.

- Ejemplo: Leyes de Kepler sobre el movimiento planetario.
  - Los planetas y el sol se reducen a puntos
  - Se consideran sólo propiedades del movimiento de los planetas (modelo cinético)
  - Se ignoran masa, composición química, etc.

# Abstracción (cont.)

## En programación

- Tenemos un lenguaje para expresar programas.
- Si deseamos que los programas sean ejecutados por computadoras, entonces deberán finalmente ser expresados en lenguaje de máquina.
- En estos lenguajes, las primitivas son elementales y por ello los programas deben ser especificados a un nivel de extremo detalle, lo cual los hace muy complejos.
- Tratamos entonces de **construir programas por refinamientos sucesivos**. Los programas se expresan en términos de componentes que no están expresados directamente en lenguaje de máquina. Luego se aplica el mismo procedimiento a cada componente, refinándolo, hasta alcanzar el nivel de detalle requerido.

# Lenguajes de programación

- Un lenguaje de programación de "alto" nivel de abstracción es aquel cuyas primitivas pueden ser refinadas de modo mecánico a lenguaje de máquina. El programa que efectúa ese refinamiento es el compilador del lenguaje.
- Ejemplo: programa que lee coordenadas de vértices de un triángulo y calcula el área de éste.  
-- Podemos expresarlo en un lenguaje de "alto nivel" como C++ de la siguiente forma:

```
< importaciones >
< declaraciones >
main () // Area Triang.
< declaraciones >
{
    < instrucciones >
} // Area Triang.
```

# Ejemplo: Area de Triángulo

- Podemos comenzar introduciendo el tipo de datos que representará el concepto de punto (en coordenadas cartesianas):

- `< declaraciones >`

```
    struct Punto {  
        float x;  
        float y; }
```

- `<...>`

- y luego las variables del programa:

- `< declaraciones ...>`

```
    Punto p1, p2, p3;
```

- `<...>`

# Ejemplo: Area de Triángulo (cont.)

- Luego refinamos la parte de instrucciones:
  - `< instrucciones >`
    - `< Leer (p1, p2, p3) >`
    - `< Desplegar (AreaTri (p1, p2, p3)) >`
    - `– < . . . >`
- Donde estamos haciendo uso de "instrucciones abstractas"
- Abstractas porque no están (todavía) dadas al nivel concreto de detalle del lenguaje de programación. Pero podemos especificar su efecto con precisión.
- Otra forma de decir lo mismo:
  - son instrucciones (se puede especificar su efecto) pero no son instrucciones concretas del lenguaje.

# Ejemplo: Area de Triángulo (cont.)

- El problema original (construir un programa) se resuelve por medio de una estructura cuyos componentes son o bien instrucciones concretas o bien otros programas a ser refinados separadamente.
  - **El método permite acotar el nivel de detalle a ser considerado cada vez.**
- Otra forma de decir lo mismo: se usan abstracciones para particionar el problema dado. Luego se refinan las abstracciones siguiendo el mismo método.

# Ejemplo: Area de Triángulo (cont.)

- En el ejemplo introducimos una función AreaTri que hay que refinar:

– < declaraciones ...>

```
float AreaTri (Punto p1, Punto p2, Punto p3)
```

```
{ < declaraciones AreaTri ...>
```

```
    < Calcular base = distancia (p1, p2) >;
```

```
    < Calcular altura = distancia p3 a p1p2 >;
```

```
    return base * altura / 2.0;
```

```
} // end AreaTri.
```



# Abstracción Procedural

- El uso de instrucciones abstractas se llama:
  - **ABSTRACCION de PROCEDIMIENTO**
  - **SUBPROGRAMAS**
  - **PROCEDIMENTAL**
  - **(Inglés: PROCEDURAL ABSTRACTION)**

y es la forma más elemental de abstracción.

- Algunas instrucciones abstractas se refinan en subprogramas (procedimientos, funciones) en C++.
- Otras simplemente en instrucciones que sustituyen textualmente a la instrucción abstracta.

# Abstracción de Datos

- Nos interesa extender la idea de abstracción a los TIPOS de DATOS.
- Es decir, usar **TIPOS ABSTRACTOS de DATOS** además de instrucciones abstractas, para diseñar programas.
- ¿Qué es un tipo abstracto de datos?

**Es un tipo de datos, es decir, puede ser especificado como tal con precisión, pero no está dado como un tipo de datos concreto del lenguaje. Esto es: no está dado en términos de los constructores de tipo del lenguaje.**

- Tipos NO abstractos en C++:
  - **elementales:** int, float, char, T \* (punteros)
  - **estructurales:** [ ] (arreglos) , struct, tipos de estructuras dinámicas.

# Abstracción de Datos (cont.)

- ¿Cómo se introduce un Tipo Abstracto de Datos?

Básicamente: dándole un nombre y asociando a él un número de operaciones aplicables a los elementos del tipo. En C++, estas operaciones son subprogramas (procedimientos, funciones, métodos).

- ¿Cómo se refina un Tipo Abstracto de Datos?

Definiéndolo como un tipo concreto del lenguaje y refinando en forma acorde los subprogramas asociados.

- Usualmente se habla de

- **ESPECIFICACION** del T.A.D.

- DEFINICION: correspondiendo a su introducción

- **IMPLEMENTACION** del T.A.D.: correspondiendo a su refinamiento.

# Abstracción de Datos (cont.)

- En este contexto, el tipo concreto provisto para definir el T.A.D. en una implementación dada se llama una **REPRESENTACION** del T.A.D.
- Hay una analogía con las estructuras algebraicas:  
Ej: monoide denomina una clase de estructuras, formadas por un conjunto, un elemento  $x$  del conjunto y una operación asociativa que tiene a  $x$  como neutro (especificación). Implementaciones:  $(N, +, 0)$ ,  $(Z, *, 1)$ ,  $(Listas, ++, [])$ ; donde  $N$ ,  $Z$  y  $Listas$  son representaciones.
- Podemos decir: un conjunto y unas operaciones que cumplen ciertas leyes.
- En general, **tipo abstracto** corresponde a “clase de estructura algebraica”, mientras **implementación** del tipo abstracto se corresponde con una “estructura concreta que satisface la especificación de la clase”.

# Uso de TADs

Veamos un caso de diseño de programa usando un tipo abstracto de datos.

**PROBLEMA:** Escribir un programa que simule una calculadora con la que se pueda operar con números racionales, dados en la forma  $m/n$  con  $m, n$  enteros ( $n \neq 0$ ).

- Pensamos en el modelo de una calculadora común, que consta de un **visor** donde se muestra el último resultado computado.
- Con ese número se puede operar, sumando, restando, multiplicando o dividiendo el mismo con otro número, lo cual da lugar a un nuevo resultado que reemplaza al anterior.
- También es posible borrar el último resultado o reemplazarlo por otro que se ingresa directamente.
- ⇒ El programa a construir aceptará comandos, posiblemente con argumentos correspondiendo a cada una de las operaciones enumeradas.
- ⇒ Contestará a cada uno de esos comandos con el resultado, simulando de esta forma el visor de la calculadora.

# Uso de TADs (cont.)

He aquí un diseño del programa:

```
< importaciones main >  
< declaraciones >  
void main() // el void es optativo  
{ // Calc. Racionales  
    < declaraciones main >  
    < instrucciones main >  
} // Calc. Racionales
```

**Comenzamos por introducir las variables (i.e. la estructura de datos del programa).**

- **Necesitamos una variable para almacenar el último resultado computado (es decir, la variable que simula el visor de la calculadora).**

# Uso de TADs (cont.)

## 1) ¿Qué tipo tendrá esa variable?

- Deberá almacenar números racionales.
- Este tipo no es un tipo de datos primitivo del lenguaje. Ahora bien, queremos continuar el diseño del programa sin detenernos a pensar en este momento cómo representar números racionales.
- El método será: usar el tipo Racional como un tipo abstracto, a ser refinado posteriormente.
- Las operaciones asociadas a este tipo abstracto serán aquellas que vayamos detectando como necesarias en el transcurso del diseño del programa.

# Definición de TADs

## 2) ¿Cómo se especificará un TAD en C++?

- Se escribe un módulo de definición, donde se introduce:
  - la definición del tipo
  - la definición de sus operaciones

pero no su implementación

O sea: en el módulo de definición sólo especificamos el tipo, no lo implementamos.

## Ventajas de este mecanismo:

- 1) Nos permite trabajar a un mayor nivel de abstracción.  
Esto es: posponer el problema de implementar este tipo.
- En general existirán varias implementaciones posibles y habrá que estudiarlas en detalle para optar por una de ellas.  
No queremos involucrarnos en ese proceso ahora.



# Definición de TADs (cont.)

2) El programa principal, como se verá, no dependerá de la implementación que oportunamente se escoja, sino sólo de la especificación del tipo.

- Por lo tanto, modificar la implementación del tipo no implica tener que modificar el programa principal.

3) Ahora refinamos el programa principal, importando el módulo de definición recién introducido.

< importaciones main >

Incluir módulo de definición del tipo Racional;

<...>

<declaraciones de main>

# Declaración de variables

- 4) Ahora podemos declarar variables de tipo Racional en el programa principal. De hecho necesitamos dos, a saber:
- una para el "visor"
  - otra para almacenar cada operando a ser combinado con el "visor" por medio de las operaciones aritméticas

< declaraciones de main >

< tipos, procedimientos, constantes de main >

**Racional visor, opnd;**

<...>

# Comandos

5) Ahora necesitamos considerar los comandos aceptados por el programa. Introducimos el tipo de los comandos como una enumeración:

< tipos, procedimientos, constantes de main >

```
enum Comando { suma, resta, producto,  
              división, borrar, ingreso, fin }
```

< ... >

y declaramos una variable de este tipo para almacenar cada comando ingresado por el usuario.

< declaraciones main >

```
Comando c;
```

< ... >

# Refinando main

6) Vamos ahora a refinar la parte de instrucciones del programa principal:

< instrucciones main >

    < inicialización >

    < bloque principal >

    < terminación >

< -- >

# Refinando el bloque principal

Iterar

LeerComando (c) ;

En el caso que c sea

    suma: < proceso suma >

|    resta: < proceso resta >

|    producto: < proceso producto >

|    división: < proceso división >

|    borrar: < proceso borrado >

|    ingreso: < proceso ingreso >

< mostrar visor si corresponde >

Mientras (c!=fin)

Ojo! , estamos usando un pseudo C++

# Refinando procesos

7) Al refinar los procesos correspondientes a los comandos, obtendremos las operaciones requeridas para el tipo Racional.

```
< proceso suma >
```

```
    LeerRacional (opnd) ;
```

```
    SumaRacional (opnd, visor) ;
```

```
< -- >
```

- Aquí **LeerRacional** lee un número racional en la variable dada como parámetro y **SumaRacional** suma el primer parámetro al segundo, modificando este último.
- La resta, producto y división funcionan de manera similar. Veamos los otros casos:

# Refinando procesos (cont.)

```
< proceso borrado >
```

```
  BorrarRacional(visor) ;
```

```
< -- >
```

```
< proceso ingreso >
```

```
  LeerRacional(visor) ;
```

```
< -- >
```

- Finalmente, mostramos el visor luego de procesado cada comando, excepto el de finalización:

```
< mostrar visor si corresponde >
```

```
  if (c!=fin) EscribirRacional(visor) ;
```

```
< -- >
```

# Operaciones del TAD Racional

- 8) La inicialización no hace otra cosa que borrar el visor y desplegar mensajes apropiados.
- Esto último es lo único que hace por su parte la terminación.
  - Es decir, que ya hemos detectado todas las operaciones que el programa principal necesita usar en relación a los racionales. Estas operaciones deberán ser importadas del módulo Racionales:

**< operaciones necesarias >**

**SumaRacional, RestaRacional,**

**ProductoRacional, DivisiónRacional,**

**BorrarRacional, LeerRacional, EscribirRacional**

**< -- >**



# Especificación de operaciones

9) Todos estos procedimientos deben ser declarados en el módulo de especificación Racionales, y sus efectos deben ser especificados con precisión:

**< declaraciones de operaciones sobre Racionales >**

**< operaciones básicas >**

```
void SumaRacional (Racional q, Racional & r);
```

```
/* precondition: q es un racional válido Q y,
```

```
   r es un racional válido R o está borrado.
```

```
   postcondición:  $r = Q + R$ , o r queda borrado si lo  
   estaba originalmente.
```

```
*/
```

y similarmente se especifica el resto de las ops.

# Operaciones básicas de Racionales

10) Las operaciones básicas mencionadas antes son:

- las que permiten formar racionales a partir de un numerador y un denominador enteros. (CONSTRUCTORAS)
  - las que permiten obtener el numerador y el denominador de un racional dado. (SELECTORAS / DESTRUCTORAS)
  - las que permiten detectar si una variable racional está o no "borrada" (definida o vacía). (PREDICADOS)
- Estas no son directamente referenciadas por nuestro programa principal, pero:
    - Serán útiles en general para otros programas que manipulen racionales.
    - Serán aquellas cuya implementación manipule directamente la representación que se elija para el tipo abstracto.

# Operaciones básicas de Racionales (cont)

- En particular, las operaciones ya declaradas arriba pueden definirse en términos de estas operaciones básicas.
- Declaramos, por lo tanto:

```
< operaciones básicas >
```

```
Racional CrearRacional (int m, int n);
```

```
/* precondition: n != 0
```

```
    retorna: el racional m/n          */
```

```
int Numerador (Racional q);
```

```
/* precondition: q es un racional válido
```

```
    retorna: el numerador de q      */
```

```
int Denominador (Racional q);
```

```
boolean EsVálido (Racional q);
```

```
< -- >
```

# Implementando el TAD Racional (cont)

- 11) En este punto, el programa principal y el módulo Racionales pueden desarrollarse en forma separada. La comunicación entre ambos ha quedado precisamente establecida. Si se compila el módulo de especificación de racionales, entonces el programa principal puede ser compilado también pues sólo depende del módulo de especificación.
- En forma independiente puede desarrollarse y compilarse el módulo de implementación de los racionales. Esto favorece el trabajo en grupo que puede operar en paralelo una vez definidos los módulos de especificación necesarios.
- 12) Por el lado del programa principal, resta solamente implementar el procedimiento de lectura de comandos y terminar de refinar el bloque principal.

# Implementando el TAD Racional (cont)

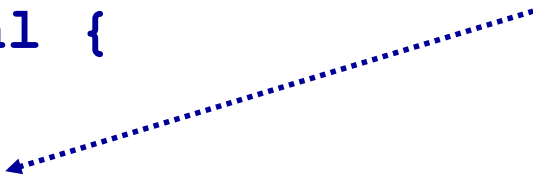
Veamos ahora cómo se puede refinar el TAD Racional.

- En nuestro caso podemos tener:

< representación elegida >

```
struct Racional {  
    int num;  
    int denom; };
```

bool válido?



< -- >


Considerar también la opción de puntero al struct (puntero NULL correspondería a que no existe el racional o que fue borrado).

# Implementando el TAD Racional (cont)

- Un ejemplo de implementación de la suma de racionales es el siguiente:

```
void SumaRacional (Racional q, Racional & r)
{ Racional sum;
  sum.num = q.num * r.denom + r.num * q.denom;
  sum.denom = q.denom * r.denom;
  r = Normalizar(sum);
}
```

Numerador(q) “**usar oper. básicas**”

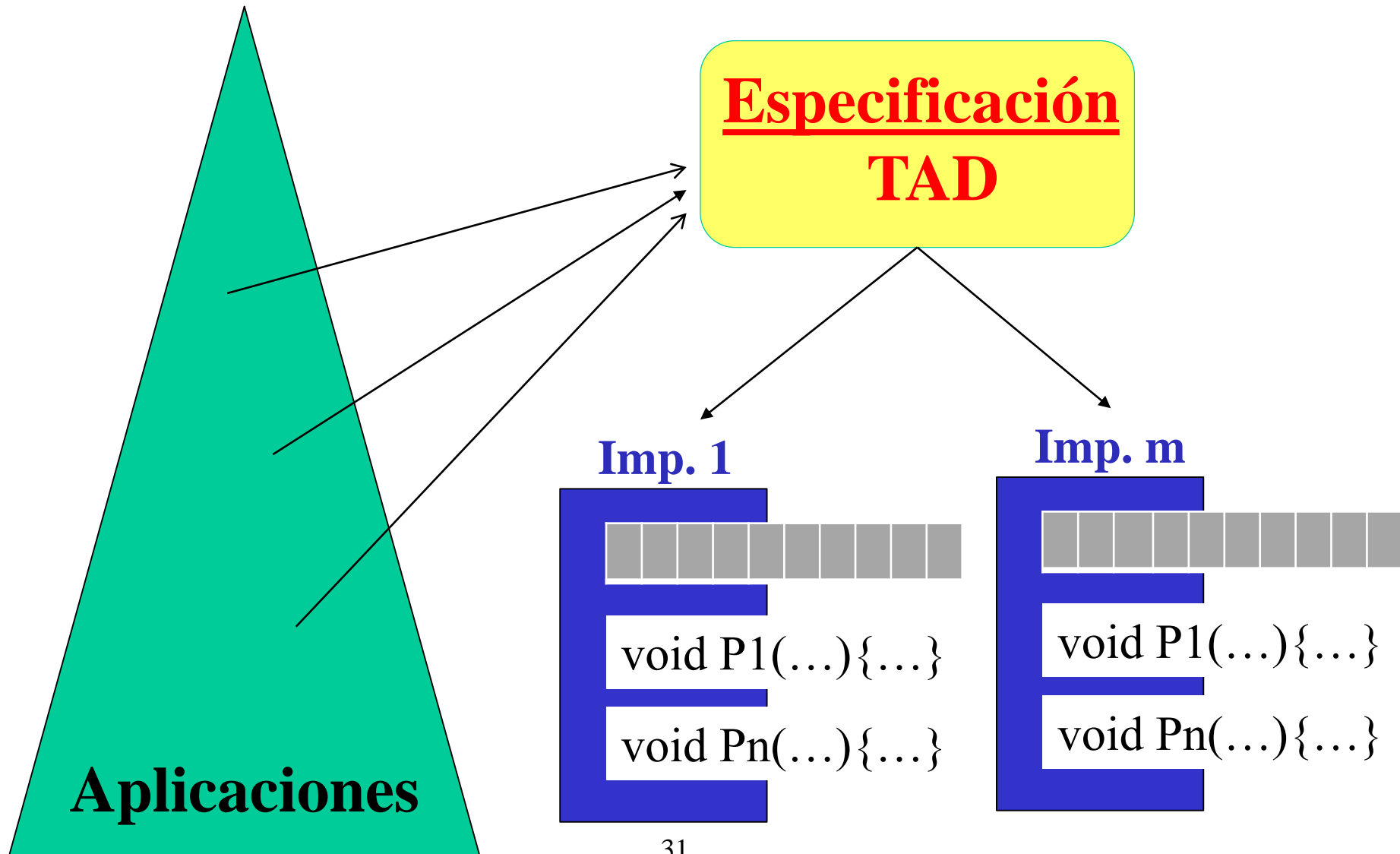


¿ **SumaRacional cumple su especificación ?**

¿ Qué pasa si *r* es (representa) a un racional borrado ? Mmm....

- El procedimiento Normalizar calcula la forma simplificada del racional dado como parámetro. Podría ser local al módulo de implementación. En general, una implementación puede tener más operaciones (auxiliares) que la especificación.

# Sobre TADs



# Conclusiones

- La metodología permite especificar tipos abstractos durante el proceso de diseño de programas.
- Provee un método conveniente de abstracción y refinamiento.
- !!! Los programas que usan tipos abstractos permanecen independientes de la implementación de éstos. Las representaciones son opacas para los programas, lo cual garantiza:
  - 1) que los programas manipulen los objetos del tipo abstracto SOLO A TRAVES DE LAS OPERACIONES DEFINIDAS (consistencia en el uso del tipo).
  - 2) que los programas no deben modificarse si la implementación del tipo abstracto se modifica.



# Conclusiones

## Algunas ventajas del uso de TADs

- Modularidad
- Adecuados para sistemas no triviales
- Separación entre especificación e implementación. Esto hace al sistema:
  - más legible
  - más fácil de mantener
  - más fácil de verificar y probar que es correcto. Robustez.
  - más fácil de reusar
  - más extensible
  - lo independiza en cierta manera de las distintas implementaciones (complejidad tiempo-espacio).

# Conclusiones

- La genericidad es una característica que buscaremos desarrollar y explotar para fomentar el reuso.
- El uso de TADs permite una rápida prototipación de sistemas, dejando de lado detalles de eficiencia para un etapa posterior.
- .....

# Observaciones

Los ejemplos desarrollados en estas transparencias hicieron uso de un pseudo C++ (pseudo-código).

Veremos en los próximos teóricos la especificación e implementación de TADs en C++. Es posible trabajar con TADs usando nociones de clases y objetos (POO). Sin embargo, también es posible trabajar con TADs en un lenguaje que no sea orientado a objetos, como C, por ejemplo. En este curso abordaremos TADs en C++ pero sin usar elementos del POO; en algunos anexos (en los próximos módulos) se ilustrará una forma de trabajar con TADs usando clases en el marco del POO.

# Ejercicio sugerido

Implementar la calculadora de racionales, siguiendo la metodología presentada en estas transparencias.

Analizar luego las implementaciones disponibles en Aulas para este problema.