

Tema 4: Tipos Abstractos de Datos

Miguel Gómez-Zamalloa Gil

Universidad Complutense de Madrid

ESTRUCTURAS DE DATOS Y ALGORITMOS
GRADO DE DESARROLLO DE VIDEOJUEGOS
CURSO 2020-2021

Bibliografía

- **Estructuras de datos y métodos algorítmicos: 213 Ejercicios resueltos.** *N. Martí Oliet, Y. Ortega Mallén y J.A. Verdejo López.* Ibergarceta Publicaciones, 2013.
- **Estructuras de datos y métodos algorítmicos: ejercicios resueltos.** *N. Martí Oliet, Y. Ortega Mallén y J.A. Verdejo López.* Pearson-Prentice Hall, 2004.
- **Estructuras de datos: un enfoque moderno.** *Mario Rodríguez Artalejo, Pedro Antonio González Calero y Marco Antonio Gómez Martín.* Editorial Complutense, 2011

Tema 7: Tipos abstractos de datos (TADS)

- Introducción
- Especificación de TADs
- Implementación de TADs en C++
- Ejemplo: El TAD Set

Introducción

Motivación

- Te plantean el siguiente problema: Dado un número x , se cogen sus dígitos y se suman sus cuadrados, para dar x_1 . Se realiza la misma operación, para dar x_2 , y así mucho rato hasta que ocurra una de las dos cosas siguientes:
 - se llega a 1, y se dice entonces que el número es “feliz”
 - nunca se llega a 1 (porque se entra en un ciclo que no incluye el 1), y se dice entonces que el número es “infeliz”
- Ejemplos:
 - el 7 es feliz: $7 \rightarrow 49 \rightarrow 97 \rightarrow 130 \rightarrow 10 \rightarrow 1$
 - el 38 es infeliz:
 $38 \rightarrow 73 \rightarrow 58 \rightarrow 89 \rightarrow 145 \rightarrow 42 \rightarrow 20 \rightarrow 4 \rightarrow 16 \rightarrow 37 \rightarrow 58$

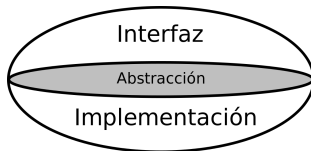
Introducción

Concepto de Abstracción

- La *abstracción* es un método de resolución de problemas
 - Los problemas tienden a ser cada vez más complejos
 - Es imposible considerar todos los detalles al mismo tiempo
- Una abstracción es un modelo simplificado de un problema donde:
 - Se consideran los aspectos de un determinado nivel
 - y se ignoran los restantes
- Ejemplos: construcción de edificio, receta albóndigas, internet, etc.
- Ventajas al razonar con abstracciones:
 - Se simplifica la resolución del problema
 - Soluciones claras \Rightarrow Razonamientos de corrección
 - Soluciones (o fragmentos de ellas) reutilizables
 - Permite la división de tareas

La Abstracción como Metodología de Programación

- La programación se beneficia del uso de abstracciones
- Evolución de la programación:
 - Código máquina
 - Ensamblador
 - Lenguajes de alto nivel (funciones, módulos, TADS, POO, prog declarativa, etc)
 - Gestores de aplicaciones (hojas de cálculo, gestores de BBDD, generadores de interfaces, etc)
- En cada nivel tenemos:



Abstracción Funcional

- Una función o procedimiento es un conjunto de sentencias que realizan una determinada acción sobre unos datos
- Una vez implementada se puede usar como si fuese una operación del lenguaje (abstracción)
- Es importante disponer de una especificación (formal o informal)

Abstracción de Datos

- El nivel de abstracción de datos ha ido aumentando:
 - Bits y bytes
 - Tipos predefinidos: `int`, `char`, `float`, etc \Rightarrow No debemos preocuparnos por su representación binaria
 - Tipos definidos por el programador. E.g. Tipo `Date`
 - TADS definidos por el programador: Entidades donde se definen conjuntamente el tipo de datos + las operaciones que los manipulan
- Observar la diferencia entre los tipos predefinidos y los definidos por el programador:
 - Un tipo predefinido es, en cierto modo, un TAD
 - Conjunto de valores + conjunto de operaciones
 - El compilador chequea que no se asignen valores erróneos
 - El programador solo necesita conocer el comportamiento de las operaciones
 - En los tipos definidos por el programador, al tener acceso a la representación interna es posible realizar operaciones incorrectas. E.g. `Tdate date; date.day = 31;`

- Un ejemplo clásico de TAD predefinido de C++ es la cadena de caracteres de su librería estándar: *std::string*
- Dada una cadena *s*, es posible saber su longitud (*s.length()* ó *s.size()*), concatenarle otras cadenas (*s += t*), sacar copias (*t = s*), o mostrarla por pantalla (*std::cout << s*), entre otras muchas operaciones
- Todo ello sin necesidad de saber cómo reserva la memoria necesaria ni cómo codifica sus caracteres

Tipos Abstractos de Datos

Un TAD se compone de:

- 1 *Interfaz*
(*visible*) \Rightarrow Especificación $\left\{ \begin{array}{l} \text{Nombre} \\ \text{Dominio del tipo} \\ \text{Operaciones y su comportamiento} \end{array} \right.$
- 2 Implementación
(*oculto*) $\left\{ \begin{array}{l} \text{Representación interna del tipo} \\ \text{Algoritmos} \Rightarrow \text{Funciones y procedimientos} \end{array} \right. \left\{ \begin{array}{l} \text{Tipos predefs} \\ \text{Tipos definidos} \\ \text{Otros TADS} \end{array} \right.$

Ejemplo: El TAD Date

1) Especificación

- Nombre: Date
- Dominio: Las fechas desde el 1/1/1
- Operaciones + especificación informal:

```
Date newDate(int day,int month,int year);// throws (Error)
    // Se devuelve un Date representando la fecha day/month/year
    // day/month/year deben representar una fecha válida,
    // sino se lanzaría una excepción

Date incr(Date date);
    // Se devuelve la fecha resultante al incrementar date en un día

int diff(Date date, Date date');
    // Se devuelve la distancia en número de días entre date y date'.

void print(Date date);
    // Imprime por consola la fecha en el formato DD/MM/YYYY
```

Ejemplo: El TAD Date

2) Implementación

- En general un TAD admite varias representaciones posibles
- Normalmente cada representación facilita unas operaciones y dificulta otras
- Alternativas para Date:
 - ❶ Registro con tres `int` \rightarrow `day`, `month` y `year`
 - ❷ Un solo `int` \rightarrow Número de días transcurridos desde el 1/1/1
- Con 1) se facilita `newDate` y `print`, y se dificulta `incr` y `diff`
- Con 2) se facilita `incr` y `diff`, y se dificulta `newDate` y `print`
- Otra alternativa sería llevar 1) + 2) ...

Soporte para TADS en Lenguajes de Programación

- Un lenguaje ofrece soporte para la implementación de TADs si incluye mecanismos para separar la interfaz de la implementación
 - Privacidad: La representación interna está oculta
 - Protección: El tipo solo puede usarse a través de sus operaciones
- **Convención:** En ocasiones se hace un “acuerdo entre caballeros” entre usuario e implementador para no tocar la parte privada
 - Ejemplo: Marca `_` en identificadores privados, escribiendo, e.g., `'_dia'`
 - Esto ya no es necesario en C++ pero sí lo era en C
- Un lenguaje soporta el uso de TADs si permite elevar su uso al rango del de los tipos predefinidos del lenguaje
- El grado de soporte varía mucho \Rightarrow Uso de módulos, POO
- En este curso usaremos la POO de C++
- Ejemplo: El TAD `Date`

Los TADs como Apoyo a la Programación Modular

- El desarrollo de programas grandes y complejos se realiza descomponiéndolos en unidades menores → *Divide y vencerás*
- Los TADs facilitan la división de tareas
- Un ejemplo:
 - Dado un vector v de N enteros y un número k , $0 \leq k \leq N$, determinar los k números mayores que aparecen en el vector
 - Usaremos una estructura auxiliar donde almacenamos los k números mayores encontrados hasta ahora. Dos opciones:
 - 1 Usar un vector (por ej) y incluir su gestión dentro del propio algoritmo
 - 2 Posponer la elección de la estructura y suponer que existe
 - Ventajas de 2):
 - 1 Descomponemos la tarea
 - 2 Reutilización del TAD resultante
 - 3 Claridad de la lógica del algoritmo

TADs y Estructuras de datos

- Un TAD está formado por una colección de valores y un conjunto de operaciones sobre dichos valores
- Una **estructura de datos** es una estrategia de almacenamiento en memoria de la información que se desea guardar
- Muchos TADs se implementan utilizando estructuras de datos
 - E.g., los TADs Pila y Cola pueden implementarse usando la estructura de datos de las listas enlazadas
- Algunos TADs son de uso tan extendido y frecuente, que es de esperar que en cualquier lenguaje de programación de importancia exista una implementación. E.g.: vectores, tablas, grafos, etc.

Especificación de TADs

Especificaciones Informales

- Las especificaciones informales son, en general, imprecisas
 - E.g. Qué ocurre si hacemos `Date date; ... date += -5; ?`
- Se podrían tratar de escribir todos los detalles textualmente ...
- Pero siempre habría ambigüedades e imprecisiones al no ser un lenguaje formal
- Aún así, las especificaciones informales son el escenario habitual
 - E.g. La *STL* de C++, la *API* de Java o la misma *JVM*

Especificaciones Formales

- Una especificación formal permite:
 - ① Verificación formal \Rightarrow Demostrar la corrección de los programas (automáticamente)
 - ② Testing formal \Rightarrow Generación automática de tests, testing automático, etc.
- Mas del 50 % de los costes en el desarrollo de software se dedica al testing!
- La verificación formal y el testing necesitan especificaciones formales
- Aspectos básicos de especificaciones formales:
 - ① Dominio del TAD \Rightarrow conjunto de posibles valores
 - ② Invariante de la representación
 - ③ Relación de equivalencia
 - ④ Especificaciones formales pre/post de cada operación

Implementación de TADs en C++

- C++ es un lenguaje tremendamente flexible y potente (incluso demasiado ...)
- Hay más de una forma de hacer las cosas
- Hay que entender las ventajas y desventajas de cada alternativa
- Veremos las “mejores prácticas” reconocidas en la industria ⇒ Referencia para hacer tus propias implementaciones

Clasificación de Operaciones

- Constructor** Crea una nueva instancia del tipo. En C++, un constructor se llama siempre como el tipo que construye. Se llaman automáticamente cuando se declara una nueva instancia del tipo.
- Mutador** Modifica la instancia actual del tipo. En C++, *no pueden* llevar el modificador *const* al final de su definición.
- Observador** No modifican la instancia actual del tipo. En C++, *deben* llevar el modificador *const* (aunque el compilador no genera errores si se omite).
- Destructor** Destruye una instancia del tipo, liberando cualquier recurso que se haya reservado en el momento de crearla (por ejemplo, cerrando ficheros, liberando memoria, o cerrando conexiones de red). Los destructores se invocan automáticamente cuando las instancias a las que se refieren salen de ámbito.

Mutadores vs. observadores

- Se debe usar el modificador **const** siempre que sea posible (i.e., para todas las operaciones observadoras)
- Hacer esta distinción tiene múltiples ventajas. Por ejemplo, una instancia de un tipo *inmutable* (sin mutadores) se puede compartir sin que haya riesgo alguno de que se modifique el original.
- Es frecuente, durante el diseño de un TAD, poder elegir entre suministrar una misma operación como mutadora o como observadora.
- E.g., en el TAD *Fecha*, podríamos elegir entre suministrar una operación *suma(int dias)* que modifique la fecha actual sumándola *delta* días (mutadora), o que devuelva una *nueva* fecha *delta* días en el futuro (observadora, y por tanto *const*).

Operaciones parciales

- Ciertas operaciones pueden ser erróneas; bien por definición o bien por limitaciones de la representación
- Ejemplos:
 - Acceso a la cima de una pila vacía
 - Acceso a un índice no válido en un vector
- Estas operaciones se denominan *parciales*
- Sus precondiciones para garantizar un comportamiento predecible deben especificarse debidamente
- Utilizaremos *excepciones* como mecanismo de tratamiento de errores

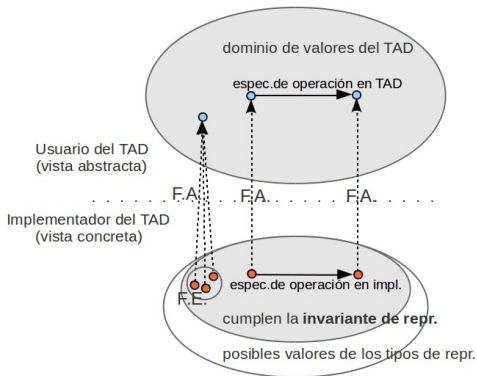
Invariante de la Representación

- Consiste en el *conjunto de condiciones que se tienen que cumplir para que una representación se considere como válida*
- Ejemplos:
 - TAD *Fecha* representado mediante tres enteros \Rightarrow Los tres enteros forman una fecha válida
 - TAD *Rectangulo* representado mediante punto origen y par de enteros para ancho y alto $\Rightarrow 0 \leq \text{_ancho} \wedge 0 \leq \text{_alto}$
- Proporcionaremos el invariante de la representación mediante lógica de primer orden (como comentario)

Relación de Equivalencia

- La **relación de equivalencia** indica cuándo dos valores del tipo implementador representan el *mismo valor* del TAD
- En C++, una forma de implementarla es sobrecargando el operador '=='

```
class Rectangulo {  
    // permite ver equivalencia de rectangulos mediante r1 == r2  
    bool operator==(const Rectangulo& r) const {  
        return (esVacio() && r.esVacio())  
            || (_alto == r._alto && _ancho == r._ancho  
                && _origen == r._origen);  
    };  
};
```



- El invariante de representación delimita los valores posibles del tipo representante *válidos*
- La función de abstracción (F.A) los pone en correspondencia con términos del TAD
- Varios valores del tipo representante pueden describir al mismo término del TAD – Función de equivalencia (F.E)

TADs Genéricos

- Un TAD genérico es aquel en el que uno o más de los tipos que se usan se dejan sin identificar, permitiendo usar las mismas operaciones y estructuras con distintos tipos concretos
- Ejemplo: TAD Conjunto
 - No debería haber grandes diferencias entre almacenar enteros, punteros arbitrarios, o Rectangulos
- Hay varias formas de conseguir esta genericidad
 - 1 Plantillas
 - 2 Herencia
 - 3 Lenguajes dinámicos

1) Plantillas

- Permite declarar tipos como “de plantilla” (*templates*), que se resuelven en tiempo de compilación para producir todas las variantes concretas que se usan realmente
- Mantienen un tipado fuerte y transparente al programador

```
// requiere <stack>, un TAD generico 'pila' via plantillas
stack<bool> s;
s.push(true);           // s.push("ey") falla
bool b = s.top();      // b == true
```

2) Herencia

- Muy usado en Java y disponible en cualquier lenguaje con soporte OO.
- Requiere que todos los tipos concretos usados descendan de un tipo base que implemente las operaciones básicas que se le van a pedir

```
// java.util.Stack (TAD generico 'pila' via herencia)
Stack s = new Stack(); // contiene Object
s.push(Boolean.FALSE); // s.top() devuelve Object
s.push("ey");           // funciona sin problemas
// Java SE 5 en adelante: version semi-generica
Stack<Boolean> t = new Stack<Boolean>();
s.push(Boolean.FALSE); // s.top() devuelve Boolean
s.push("ey");           // error de compilacion
```

3) Lenguajes dinámicos

- JavaScript o Python son lenguajes que permiten a los tipos adquirir o cambiar sus operaciones en tiempo de ejecución
- En estos casos, basta con que los objetos de los tipos introducidos soporten las operaciones requeridas en tiempo de ejecución (pero se pierde la comprobación de tipos en compilación)

Plantillas en C++

- En C++, se pueden definir TADs genéricos usando la sintaxis

template <**class** T_1 , ... **class** T_n > *contexto*

y refiriéndose a los T_i igual que se haría con cualquier otro tipo a partir de este momento

- Generalmente se escogen mayúsculas que hacen referencia a su uso
- Por ejemplo, para un tipo cualquiera se usaría T , para un elemento E ; etc.

Plantillas en C++

Ejemplo de TAD genérico: Pair

```
// en el .h
template <class A, class B>
class Pair {
    // una pareja inmutable generica
    A _a; B _b;
public:
    // Generador (un constructor sencillo)
    Pair(A a, B b) { _a=a; _b=b; } // cuerpos en el .h
    // Observadores
    //   Pair(a, b).first() = a
    A first() const { return _a; }
    //   Pair(a, b).second() = b
    B second() const { return _b; }
};
```

Plantillas en C++

```
// en el main.cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    Pair<int, string> p(4, "hola");
    cout << p.first() << " " << p.second() << "\n";
    return 0;
}
```

Plantillas en C++

- En C++ se pueden definir los cuerpos de funciones en el .h en lugar de en el prototipo (tal y como se hace en el ejemplo)
- En el caso de TADs genéricos, esto es **obligatorio** (en caso contrario se producirán errores de enlazado)
- Aunque para implementaciones más grandes es preferible usar esta versión alternativa, que deja el tipo más despejado, a costa de repetir la declaración de los tipos de la plantilla para cada contexto en el que se usa:

Plantillas en C++

```
// en el. h
...
// Generador (un constructor sencillo)
Pair(A a, B b);
// Observadores
A first() const;
B second() const;
};

template <class A, class B>
Pair<A,B>::Pair(A a, B b) { _a = a; _b = b; }

template <class A, class B>
A Pair<A,B>::first() const { return _a; }

template <class A, class B>
B Pair<A,B>::second() const { return _b; }
```

Ejemplo: El TAD Set

```
template <class E>
class Set {
    ... // Tipo representante
public:
    // Constructor
    Set();

    // Inserta un elemento (mutadora)
    // Podría dar error si no hay espacio
    void add(const E &e);

    // Elimina un elemento (mutadora)
    void remove(const E &e);

    // Si contiene e, devuelve 'true' (observadora)
    bool contains(const E &e) const;
};
```
