

Implementación y uso de TADs

Manuel Freire es el autor principal de este tema.
Modificado por Clara Segura. Curso 13-14

Facultad de Informática - UCM

17 de febrero de 2014

Tipos abstractos de datos = conjunto de valores + operaciones.

TADs predefinidos:

Valores	Representación	Operaciones
bool	un bit	<i>true, false</i> , \neg , \wedge , \vee
int	complemento a 2	$+$, $-$, $*$, $/$, $\%$, $==$, $!=$, $<$...
float	mantisa y exponente	$+$, $-$, $*$, $/$, $==$, $!=$, $<$...
char	1 byte	$+$, $-$, $<$, $>$, $==$, ...
array	puntero	$-[]$,
ficheros	lectura, escritura...

- Cuando se realiza una abstracción que no permite acceso a la implementación subyacente, se sacrifican algunas optimizaciones.
- Aproximación al cálculo de la inversa de la raíz cuadrada

```
float Q_rsqrt(float n) {  
    const float threehalfs = 1.5f;  
    float x2 = n * 0.5f;  
    float y = n;  
    // convierte de float a long (!)  
    int i = * ( long * ) &y;  
    i = 0x5f3759df - ( i >> 1 );  
    // convierte de long a float (!)  
    y = * ( float * ) &i;  
    y = y * ( threehalfs - ( x2 * y * y ) );  
    return y;  
}
```

- **struct** define nuevos tipos de datos, pero no ocultan los detalles de la implementación.
- **programación OO / clases** permite ocultar los detalles de la implementación de los tipos.
- Ejemplo:

```
struct Fecha {  
    int dia;      int mes;      int anyo;  
    Fecha(int d, int m, int a):  
        dia(d), mes(m), anyo(a) {}  
};
```

- Crear una fecha: `Fecha f = Fecha(14, 7, 1789)`
- Otra fecha: `Fecha f(14, 7, 1789),`
- Acceder a un campo: `f.dia`
- Es posible crear fechas inconsistentes; `f.mes = 13`
- No incluye operaciones para sumar o restar días a una fecha, ni para calcular en qué día de la semana cae una fecha dada, etc.

```
class Fecha {  
public: // Interfaz  
    // Constructor (generador)  
    //   dia, mes y año forman una fecha  
    Fecha(int dia, int mes, int anyo);  
  
    // Constructor alternativo (generador)  
    // dias a sumar o restar a la fecha base  
    Fecha(const Fecha &base, int dias);  
  
    // distancia, en dias, de la fecha actual con la dada  
    int distancia(const Fecha &otra) const;  
  
    // devuelve el dia de la semana de esta fecha  
    int diaSemana() const;
```

```
// devuelve el día de esta fecha  
int dia() const;
```

```
// devuelve el mes de esta fecha  
int mes() const;
```

```
// devuelve el año de esta fecha  
int anyo() const;
```

private: accesible solo para la implementacion

```
int _dia;  
int _mes;  
int _anyo;
```

```
};
```

```
// Se pueden definir operaciones privadas que ayuden  
// en la implementación de las funciones publicas
```

```
Fecha::Fecha(int dia, int mes, int anyo) {
    _dia = dia; _mes = mes; _anyo = anyo;
}
int Fecha::dia() const { return _dia; }

int main() {
    Fecha f(14, 7, 1789);
    std::cout << "para f(14, 7, 1789), f.dia() = "
                << f.dia() << "\n";

    return 0;
}
```

- Podríamos haber elegido una implementación que almacenase sólo `_diaAnyo` y `_anyo`.
- Facilitaría la implementación de `distancia()` entre fechas del mismo año la implementación de `diaAnyo()`,
- Complicaría `dia()` ó `mes()`.

Implementación de un TADs en C++

- La *declaración* de una clase se escribe en un fichero .h (por ejemplo, fecha.h),

```
// includes imprescindibles para que compile el .h
#include <libreria_sistema>

// protección contra inclusión múltiple
#ifndef _FECHA_H_
#define _FECHA_H_

// ... declaración de la clase Fecha aqui ...

// fin de la protección contra inclusión múltiple
#endif
```

Implementación de un TADs en C++

- La *implementación* en un fichero .cpp asociado (por ejemplo, fecha.cpp):

```
#include "fecha.h"
```

```
// includes adicionales para que compile el .cpp  
#include <libreria_sistema>
```

```
// implementación de las operaciones de Fecha ...
```

- Se puede declarar más de una clase en un fichero .h

Tipos de operaciones y el modificador *const*

- Constructor/generador.
 - Crea una nueva instancia del tipo.
 - Se llama siempre como el tipo que construye (con los argumentos que se desee utilizar).
 - Se llaman automáticamente cuando se declara una nueva instancia del tipo.
 - Si no hay nada que inicializar, se pueden omitir.
- Mutador/modificador. Modifica la instancia actual del tipo.
- Observador.
 - No modifican la instancia actual del tipo.
 - *Deben* llevar el modificador `const` (aunque el compilador no genera errores si se omite).

Tipos de operaciones y el modificador *const*

- **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.
 - Si no hay que liberar nada, se pueden omitir.
- Es frecuente, poder elegir entre suministrar una misma operación como mutadora o como observadora. Por ejemplo, en `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*).

Uso de sub-TADs y extensión de TADs

- Un TAD puede usar a otro. Ejemplo un TAD Rectangulo puede usar un Punto:

```
#include "Punto.h" // usa Punto
// Un Rectangulo en 2D alineado con los ejes.
class Rect {
private:
    ...
public:
    // const. origen es la esquina inf. izquierda
    Rect(Punto origen, float alto, float ancho);
    // const. interpreta ambos puntos como esquinas
    Rect(Punto uno, Punto otro);
    // dev punto con coord x e y mínimas (Obs.)
    const Punto &origen() const;
    float alto() const;
    float ancho() const;
    ...
};
```

TADs genéricos

- 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`, las operaciones no expenden del tipo de los elementos.
- Hay varias formas de conseguir esta genericidad:
 - plantillas: usado en C++; 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.
 - herencia: 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. Tienen mayor coste que los templates.
 - lenguajes dinámicos: JavaScript o Python son lenguajes que permiten a los tipos adquirir o cambiar sus operaciones en tiempo de ejecución.

TADs genéricos

Sintaxis C++

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

Ejemplo de TAD genérico Pareja:

```
template <class A, class B>
class Pareja {
    A _a; B _b;
public:
    // Constructor
    Pareja(A a, B b) { _a=a; _b=b; } // cuerpos en el .h
    // Observadoras
    A primero() const { return _a; }
    B segundo() const { return _b; }
};
```

TADs genéricos

```
#include <iostream>
#include <string>
#include "pareja.h"
using namespace std;
int main() {
    Pareja<int,string> p(4, "hola");
    cout << p.primer() << " " << p.segundo() << "\n";
    return 0;
}
```

TADs genéricos

- En los TADs genéricos es **obligatorio** implementar las operaciones en el fichero .h (en caso contrario se producirán errores de enlazado).
- Si la implementación es extensa realizarla detrás de la declaración de la clase.

```
template <class A, class B>
class Pareja {
    ...
public:
    ...
};

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

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

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

Definición & Implementación de un TAD

- Un TAD se puede implementar de diversas formas. Cada implementación tendrá unas limitaciones diferentes y un coste diferente para sus operaciones.
- Pasos para implementar un TAD:
 - ❶ Elegir los tipos de implementación concretos que se va a usar para representar el TAD (**tipos representantes**);
 - ❷ Dar la interpretación que se va a hacer de sus valores (**función de abstracción**,
 - ❸ Decidir cuándo dos valores del tipo representante representan lo mismo: (**función de equivalencia**);
 - ❹ Definir las condiciones que se deben cumplir para que los valores se consideren válidos (**invariantes de representación**).
 - ❺ Implementar las operaciones, de forma que nunca se rompan los invariantes de representación (posiblemente con restricciones adicionales debidas al tipo representante elegido).

TAD Rectangulo en 2D alineado con los ejes.

Primera representación

- **Tipos.** Un `Punto` y un par de `int`.
- **Función de abstracción:** El `Punto` representa el extremo inferior izquierdo del rectángulo. Los valores enteros el ancho y alto. Un rectángulo es vacío si $\text{alto} = \text{ancho} = 0$.
- **Función de equivalencia.** Dos rectángulos son iguales cuando tiene el mismo origen y el mismo alto y ancho. Los rectángulos vacíos se consideran iguales.
- **Invariante de la representación.** El ancho y alto deben ser valores mayores o iguales que cero.

TAD Rectángulo en 2D alineado con los ejes.

Segunda representación

- **Tipos.** Dos puntos.
- **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.
- **Función de equivalencia.** Dos rectángulos son iguales cuando coinciden sus puntos.
- **Invariante de la representación.** Las coordenadas x e y del segundo punto deben ser mayores que las del primero.

Definición & Implementación de un TAD

TAD Fecha.

Primera representación

- **Tipos.** Tres enteros.
- **Función de abstracción:** Los enteros almacenan los días, meses y años. Los años se representan únicamente con dos cifras.
- **Función de equivalencia.** Dos fechas son iguales cuando coinciden sus valores.
- **Invariante de la representación.** Se deben respetar las reglas de construcción de fechas, meses de 30 días etc.

Definición & Implementación de un TAD

TAD Fecha.

Segunda representación

- **Tipos.** Un entero.
- **Función de abstracción:** El entero representa los días transcurridos desde el 1 de enero de 1970.
- **Función de equivalencia.** Dos fechas son iguales cuando coinciden su valor.
- **Invariante de la representación.** Cualquier valor positivo o negativo.

Definición & Implementación de un TAD

Restricciones impuestas por el tipo representante al dominio de valores del TAD:

- Representar el año mediante dos caracteres “XX”, interpretándolo como “19XX”. Esto ocasionó numerosos problemas cuando se llegó al año 2000, ya que estos sistemas interpretaban como si fuese 1900.
- Un valor entero representado mediante un `int` de 32 bits sólo puede tomar 2^{32} valores distintos. Si representamos una Fecha con un `int` para los segundos desde 1970, será imposible producir fechas más allá del 19 de enero de 2038.

Esta representación es muy popular, y aunque muchas implementaciones ya se han actualizado a enteros de 64 bits, el problema es comparable al del año 2000. Para más referencias, ver [http:](http://en.wikipedia.org/wiki/Year_2038_problem)

[//en.wikipedia.org/wiki/Year_2038_problem](http://en.wikipedia.org/wiki/Year_2038_problem).

Implementar la relación de equivalencia

En C++, la relación de equivalencia se implementa sobrecargando el operador '==':


```
class Rectangulo {  
    ...  
    // 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);  
    };  
    ...  
};
```

TAD Conjunto

Tipo representante: Vector de tamaño fijo, con el índice del ultimo elemento indicado mediante un entero `_tam`:

```
template <class E>
class Conjunto {
    static const int MAX = 100;
    int _tam = 0;
    E _elementos[MAX];
public:
    // Constructor
    Conjunto();
    // inserta un elemento (mutadora)
    void inserta(const E &e);
    // elimina un elemento (mutadora)
    //     parcial: si no contiene(e), error
    void elimina(const E &e);
    // si contiene el elemento, devuelve 'true' (obs.)
    bool contiene(const E &e) const;
};
```


TAD Conjunto

- **Función de abstracción:** Los elementos con índice 0 a `_tam` (exclusive) forman parte del conjunto (salvo posibles repeticiones).
 - **Relación de equivalencia.** Múltiples vectores pueden referirse al mismo conjunto abstracto:
 - A partir de `_tam` (inclusive), el contenido del vector `_elementos` es completamente indiferente desde el punto de vista del TAD.
 - Las repeticiones y el orden son irrelevantes.
-  la basura, no los datos

- **Invariante de representación:** $0 \leq _tam \leq MAX$ (siempre se deben usar constantes¹ para los valores límite). Podemos considerar tres invariantes de representación diferentes.:
 - ❶ No hay ninguna restricción adicional a lo que se ha dicho hasta el momento.
 - ❷ Exigimos que los elementos del vector entre 0 y $_tam - 1$ no estén repetidos.
 - ❸ Exigimos que los elementos del vector entre 0 y $_tam - 1$ no estén repetidos y además estén ordenados (supuesto que exista una relación de orden entre ellos).

¹El uso de `static const` declara `MAX` como constante (`const`) para esta implementación del TAD, definida una única vez para cualquier número de conjuntos (`static`), en lugar de una vez para cada conjunto individual (ausencia de `static`). Es preferible usar constantes estáticas de clase que `#defines`.

Implementaciones de cada operación, respetando los invariantes.

Representación con invariante (1):

```
Conjunto() { _tam = 0; }
```

La operación de insertar es *parcial*.

```
void inserta(const E &e) {  
    if (_tam == MAX) throw "Conjunto Lleno";  
    _elementos[_tam] = e;  
    _tam ++;  
}
```

TAD Conjunto

Para implementar la operación de eliminación, hemos de tener en cuenta que puesto que los elementos pueden estar repetidos, hay que eliminar todas las apariciones de dicho elemento. La operación es *parcial* si el elemento no está se produce un error:

```
void elimina(const E &e) {
    bool esta = false;
    for (int i=0;i<_tam;i++)
        { if (_elementos[i]==e) {
            _elementos[i] = _elementos[_tam-1];
            esta = true;
            _tam --;
            // paso el ultimo elemento a su lugar
        };
    if (!esta) throw "Elemento Invalido";

}
```

TAD Conjunto

```
bool contiene(const E &e) const {  
    bool esta = false;  
    int i=0;  
    while (i<_tam && !esta)  
        {esta = (_elementos[i]==e);  
          i++;  
        };  
    return esta;  
}
```

Con esta forma de representar los conjuntos, las operaciones tienen los siguientes costes:

- El constructor y la operación `inserta` están en $\Theta(1)$.
- Las operaciones `elimina` y `contiene` están en $\Theta(_tam)$ en el caso peor.

Obsérvese que puesto que puede haber elementos repetidos, `_tam` puede ser mucho más grande que el cardinal del conjunto al que representa. En las otras representaciones, que exigen que no haya repetición de elementos, `_tam` representa exactamente el cardinal del conjunto.