

CAPÍTULO 17

Tipos abstractos de datos TAD/objetos

En este capítulo se examinan los conceptos de *modularidad, abstracción de datos* y *objetos*. La modularidad es la posibilidad de dividir una aplicación en piezas más pequeñas llamadas módulos. *Abstracción de datos* es la técnica de inventar nuevos tipos de datos que sean más adecuados a una aplicación y, por consiguiente, facilitar la escritura del programa. La técnica de abstracción de datos es una técnica potente de propósito general que cuando se utiliza adecuadamente, puede producir programas más cortos, más legibles y flexibles. Los *objetos* combinan en una sola unidad *datos* y *funciones* que operan sobre esos datos.

Los lenguajes de programación soportan en sus compiladores *tipos de datos fundamentales o básicos (predefinidos)*, tales como `int`, `char` y `float` en C. Casi todos los lenguajes de programación tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos.

Un tipo de dato definido por el programador se denomina *tipo abstracto de dato, TAD, (abstract data type, ADT)*. El término abstracto se refiere al medio en que un programador abstrae algunos conceptos de programación creando un nuevo tipo de dato.

La modularización de un programa utiliza la noción de *tipo abstracto de dato (TAD)* siempre que sea posible. Si el lenguaje de programación soporta los tipos que desea el usuario y el conjunto de operaciones sobre cada tipo, se obtiene un nuevo tipo de dato denominado *TAD*.

17.1 Tipos de datos

Todos los lenguajes de programación soportan algún tipo de datos. Por ejemplo, el lenguaje de programación convencional C soporta tipos base tales como enteros, reales y caracteres; así como tipos compuestos tales como *arrays* (vectores y matrices) y *estructuras*(registros).

A tener en cuenta

Un tipo de dato es un conjunto de valores, y un conjunto de operaciones definidas sobre esos valores.

Un valor depende de su representación y de la interpretación de la representación, por lo que una definición informal de un tipo de dato es: *Representación + Operaciones*.

Un *tipo de dato* describe un conjunto de objetos con la misma representación. Existen un número de operaciones asociadas con cada tipo. Es posible realizar aritmética sobre tipos de datos enteros y reales, concatenar cadenas o recuperar o modificar el valor de un elemento.

La mayoría de los lenguajes tratan las variables y constantes de un programa como *instancias* de un *tipo de dato*. Un tipo de dato proporciona una descripción de sus instancias que indica al compilador cosas como cuánta memoria se debe asignar para una instancia, cómo interpretar los datos en memoria y qué operaciones son permisibles sobre esos datos. Por ejemplo, cuando se escribe una declaración tal como `float z` en C ó C++, se está declarando una instancia denominada `z` del tipo de dato `float`. El tipo de datos `float` indica al compilador que reserve, por ejemplo, 32 bits de memoria, y qué operaciones tales como “*sumar*” y “*multiplicar*” están permitidas, mientras que operaciones tales como el “*el resto*” (*módulo*) y “*desplazamiento de bits*” no lo están. Sin embargo, no se necesita escribir la declaración del tipo `float` -el autor de compilador lo hizo ya y se construyen en el compilador-. Los tipos de datos que se construyen en un compilador de este modo, se conocen como *tipos de datos fundamentales (predefinidos)*, y por ejemplo en C y C++ son entre otros: `int`, `char`, `float` y `double`.

Cada lenguaje de programación incorpora una colección de tipos de datos fundamentales, que incluyen normalmente enteros, reales, carácter, etc. Los lenguajes de programación soportan también un número de constructores de tipos incorporados que permiten generar tipos más complejos. Por ejemplo, C soporta registros (estructuras) y arrays.

A tener en cuenta

El programador no tiene que preocuparse de saber cómo el compilador del lenguaje implementa los tipos de datos predefinidos, simplemente usa los tipos de datos en el programa.

17.2 Tipos abstractos de datos

Algunos lenguajes de programación tienen características que permiten ampliar el lenguaje añadiendo sus propios tipos de datos. Un tipo de dato definido por el programador se denomina *tipo abstracto de datos (TAD)*, se implementa considerando los valores que se almacenan en las variables y las operaciones disponibles para manipular esas variables. Por ejemplo, en C el tipo *Punto*, que representa a las coordenadas *x* e *y* de un sistema de coordenadas rectangulares, no existe; el programador puede definir el *tipo abstracto de datos Punto* que represente las coordenadas rectangulares, y las operaciones que se pueden realizar (*distancia*, *módulo* ...). En esencia un tipo abstracto de datos es un tipo de datos que consta de datos (estructuras de datos propias) y operaciones que se pueden realizar sobre esos datos. Un **TAD** se compone de *estructuras de datos* y los *procedimientos o funciones* que manipulan esas estructuras de datos.

Para recordar

Un tipo abstracto de datos puede definirse mediante la ecuación:

$$\text{TAD} = \text{Representación (datos)} + \text{Operaciones (funciones y procedimientos)}$$

Desde un punto de vista global, un *tipo abstracto de datos* se compone de la interfaz y de la implementación (Figura 17.1). Las estructuras de datos reales elegidas para almacenar la representación de un tipo abstracto de datos son invisibles a los usuarios o clientes. Los algoritmos utilizados para implementar cada una de las operaciones de los TAD están encapsuladas dentro de los propios TAD. La característica de ocultamiento de la información del TAD significa que disponen de *interfaces públicas*, sin embargo, las representaciones e implementaciones de esas interfaces son *privadas*.

VENTAJAS DE LOS TIPOS ABSTRACTOS DE DATOS

Un *tipo abstracto de datos* es un modelo (estructura) con un número de operaciones que afectan a ese modelo. Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se pueden resumir en los siguientes:

1. Permite una mejor conceptualización y modelización del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.
2. Mejora la robustez del sistema. Los tipos abstractos de datos permiten la comprobación de tipos para evitar errores de tipo en tiempo de ejecución.

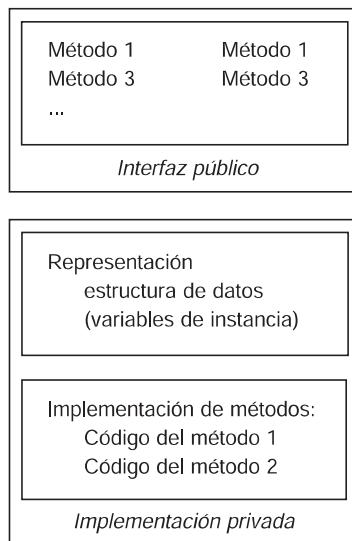


Figura 17.1 Estructura de un tipo abstracto de datos (TAD)

3. Mejora el rendimiento (prestaciones). Para sistemas tipificados, el conocimiento de los objetos permite la optimización de tiempo de compilación.
4. Separa la implementación de la especificación. Permite la modificación y mejora de la implementación sin afectar a la interfaz pública del tipo abstracto de dato.
5. Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
6. Recoge mejor la semántica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representación de atributos.

Un programa que maneja un TAD lo hace teniendo en cuenta las operaciones o funcionalidad que tiene, sin interesarse por la representación física de los datos. Es decir, los *usuarios* de un TAD se comunican con este a partir de la interfaz que ofrece el TAD mediante funciones de acceso. Podría cambiarse la implementación de tipo de datos sin afectar al programa que usa el TAD ya que para el programa está *oculta* la implementación.

IMPLEMENTACIÓN DE LOS TAD

Los lenguajes convencionales, tales como C, permiten la definición de nuevos tipos y la declaración de funciones para realizar operaciones sobre objetos de los tipos. Sin embargo, tales lenguajes no permiten que los datos y las operaciones asociadas sean declaradas juntas como una unidad y con un solo nombre. En los lenguajes en los que los módulos (TAD) se pueden implementar como una unidad, éstos reciben nombres distintos:

Turbo Pascal	<i>unidad, objeto</i>
Modula-2	<i>módulo</i>
Ada	<i>paquete</i>
C++	<i>clase</i>
Java	<i>clase</i>

En estos lenguajes se definen la *especificación* del TAD, que declara las operaciones y los datos ocultos al exterior, y la *implementación*, que muestra el código fuente de las operaciones y que permanece oculto al exterior del módulo.

En C no existe como tal una construcción del lenguaje para especificar un TAD. Sin embargo se puede agrupar la interfaz y la representación de los datos en un archivo de inclusión: *archivo.h*. La implementación de la interfaz, de las funciones se realiza en el correspondiente *archivo.c*. Los detalles de la codificación de las funciones quedan *ocultos* en el *archivo.c*.

Las ventajas de los TAD se pueden manifestar en toda su potencia, debido a que las dos partes de los módulos (*especificación* e *implementación*) se pueden compilar por separado mediante la técnica de compilación separada (“*separate compilation*”).

17.3 Especificación de los TAD

Un tipo abstracto de datos es un tipo de datos definido por el usuario que tiene un conjunto de datos y unas operaciones. La especificación de un TAD consta de dos partes, la descripción matemática del conjunto de datos, y las operaciones definidas en ciertos elementos de ese conjunto de datos. El objetivo de la especificación es describir el comportamiento del TAD.

La especificación del TAD puede tener un enfoque *informal*, en el que se describen los datos y las operaciones relacionadas en *lenguaje natural*. Otro enfoque más riguroso, especificación formal, supone suministrar un conjunto de axiomas que describen las operaciones en su aspecto sintáctico y semántico.

PROBLEMAS RESUELTOS

- 17.1.** Realizar una especificación informal del TAD *Conjunto* con las operaciones: *ConjuntoVacio*, *Esvacio*, *Añadir* un elemento al conjunto, *Pertenece* un elemento al conjunto, *Retirar* un elemento del conjunto, *Union* de dos conjuntos, *Intersección* de dos conjuntos e *Inclusión* de conjuntos.

Análisis del problema

La especificación informal consiste en dos partes:

- detallar en los datos del tipo, los valores que pueden tomar.
- describir las operaciones, relacionándolas con los datos.

El formato que especificación emplea, primero especifica el nombre del TAD y los datos:
TAD *nombre del tipo* (valores y su descripción)

A continuación cada una de las operaciones con sus argumentos, y una descripción funcional en lenguaje natural.
Operación(argumentos). Descripción funcional

Como ejemplo se va a especificar el tipo abstracto de datos *Conjunto*:

TAD Conjunto (Especificación de elementos sin duplicidades pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).

Operaciones, existen numerosas operaciones matemáticas sobre conjuntos, algunas de ellas:

Conjuntovacio.

Crea un conjunto sin elementos

Añadir(Conjunto, elemento).

Comprueba si el elemento forma parte del conjunto, en caso negativo es añadido. La función modifica al conjunto.

Retirar(Conjunto, elemento).

En el caso de que el elemento pertenezca al conjunto es eliminado de este. La función modifica al conjunto.

Pertenece(Conjunto, elemento).

Verifica si el elemento forma parte del conjunto, en cuyo caso devuelve *cierto*.

Esvacio(Conjunto)

Verifica si el conjunto no tiene elementos, en cuyo caso devuelve *cierto*.

Cardinal(Conjunto)

Devuelve el número de elementos del conjunto

Union (Conjunto, Conjunto).

Realiza la operación matemática de la unión de dos conjuntos. La operación devuelve un conjunto con los elementos comunes y no comunes a los dos argumentos.

Intersección (Conjunto, Conjunto).

Realiza la inclusión matemática de la intersección de dos conjuntos. La operación devuelve un conjunto con los elementos comunes a los dos argumentos.

Inclusión (Conjunto, Conjunto).

Verifica si el primer conjunto está incluido en el conjunto especificado en el segundo argumento, en cuyo caso devuelve *cierto*.

- 17.2. Realizar la especificación formal del TAD Conjunto con las operaciones indicadas en el ejercicio 17.1. Considerar a las operaciones ConjuntoVacio y Añadir como constructores.**

Análisis del problema

La especificación formal proporciona un conjunto de axiomas que describen el comportamiento de todas las operaciones. La descripción ha de incluir un parte de sintaxis, en cuanto a los tipos de los argumentos y el tipo del resultado, y una parte de semántica, donde se detalla para unos valores particulares de los argumentos la expresión del resultado que se obtiene. La especificación formal ha de ser lo bastante *potente* para que cumpla el objetivo de verificar la corrección de la implementación del TAD.

El esquema que se sigue para especificar formalmente un TAD consta de una cabecera con el nombre del TAD y los datos: TAD *nombre del tipo* (valores que toma los datos del tipo)

Le sigue la sintaxis de las operaciones (se listan las operaciones indicando los tipos de los argumentos y el tipo del resultado):

Sintaxis

Operación(Tipo argumento, ...)-> Tipo resultado

y a continuación, la semántica de las operaciones. Esta se construye dando unos valores particulares a los argumentos de las operaciones, a partir de los cuales se obtiene una expresión resultado que puede tener referencias a tipos ya definidos, valores de tipo lógico o referencias a otras operaciones del propio TAD.

Semántica

Operación(valores particulares argumentos) \Rightarrow expresión resultado

Al hacer una especificación formal siempre hay operaciones definidas por sí mismas, se consideran constructores del TAD. Se puede decir que mediante estos constructores se generan todos los posibles valores del TAD. Normalmente, se elige como constructor la operación que inicializa (por ejemplo, *Conjuntovacío* en el TAD Conjunto), y la operación que añade un dato o elemento (esta operación es común a la mayoría de los tipos abstractos de datos). Se acostumbra a marcar con un asterisco a las operaciones que son constructores.

A continuación se hace la especificación formal del TAD Conjunto, para formar la expresión resultado se hace uso, si es necesario, de la sentencia alternativa *si-entonces-sino*.

TAD Conjunto(colección de elementos sin duplicidades, pueden estar en cualquier orden, se usa para representar los conjuntos matemáticos con sus operaciones).

Sintaxis

*Conjuntovacío	->	Conjunto
*Añadir(Conjunto, Elemento)	->	Conjunto
Retirar(Conjunto, Elemento)	->	Conjunto
Pertenece(Conjunto, Elemento)	->	Conjunto
Esvacio(Conjunto)	->	boolean
Cardinal(Conjunto)	->	entero
Union(Conjunto, Conjunto)	->	Conjunto
Interseccion(Conjunto, Conjunto)	->	Conjunto
Incluido(Conjunto, Conjunto)	->	boolean
Semántica		$\forall e1, e2 \in \text{Elemento} \text{ y } \forall C, D \in \text{Conjunto}$

Añadir(Añadir(C, e1), e1)	\Rightarrow Añadir(C, e1)
Añadir(Añadir(C, e1), e2)	\Rightarrow Añadir(Añadir(C, e2), e1)
Retirar(Conjuntovacio, e1)	\Rightarrow Conjuntovacio
Retirar(Añadir(C, e1), e2)	\Rightarrow si e1 = e2 entonces Retirar(C,e2) sino Añadir(Retirar(C,e2),e1)
Pertenece(Conjuntovacio, e1)	\Rightarrow falso
Pertenece(Añadir(C, e2), e1)	\Rightarrow si e1 = e2 entonces cierto sino Pertenece(C, e1)
Esvacio(Conjuntovacio)	\Rightarrow cierto
Esvacio(Añadir(C, e1))	\Rightarrow falso
Cardinal(Conjuntovacio)	\Rightarrow Cero
Cardinal(Añadir(C, e1))	\Rightarrow si Pertenece(C,e1) entonces Cardinal(C) sino 1 + Cardinal(C)
Union(Conjuntovacio, Conjuntovacio)	\Rightarrow Conjuntovacio
Union(Conjuntovacio, Añadir(C, e1))	\Rightarrow Añadir(C, e1)
Union(Añadir(C, e1), D)	\Rightarrow Añadir(Union(C, D), e1)
Interseccion(Conjuntovacio, Conjuntovacio)	\Rightarrow Conjuntovacio
Intereseccion(Añadir(C, e1),Añadir(D, e1))	\Rightarrow Añadir(Interseccion (C,D), e1)
Incluido(Conjuntovacio, Conjuntovacio)	\Rightarrow cierto
Incluido(Añadir(C, e1), Añadir(D, e1))	\Rightarrow cierto si Incluido (C, D)

- 17.3. Crear un TAD que represente un dato tipo cadena (string) y sus diversas operaciones: *CadenaVacia*, *Asignar*, *Longitud*, *Buscar posición de un carácter dado*, *Concatenar cadenas*, *Extraer una subcadena*. Realizar la especificación informal y formal considerando como constructores las operaciones *CadenaVacia* y *Asignar*.

ESPECIFICACIÓN INFORMAL

TAD Cadena (Secuencia de caracteres ASCII terminada por un byte nulo).

Operaciones

Cadenavacia.

Crea una cadena vacía

Asignar (Cadena, Cadena1).

Elimina el contenido de la primera cadena si lo hubiere y lo sustituye por la segunda.

Longitud (Cadena).

Devuelve el número de caracteres de la cadena sin contar el byte final.

Buscar (Cadena, Carácter)

Devuelve la posición de la primera ocurrencia del carácter por la izquierda.

Concatenar (Cadena1, Cadena2).

Añade el contenido de Cadena2 a la cadena del primer argumento.

Extraer (Cadena, Posición, NumCaracteres).

Devuelve la subcadena del primer argumento que comienza en la posición del segundo argumento y tiene tantos caracteres como indica el tercero.

ESPECIFICACIÓN FORMAL

TAD Cadena (Secuencia de caracteres ASCII terminada por un byte nulo).

Sintaxis

*Cadenavacia	-> Cadena
*Asignar (Cadena, Cadena)	-> Cadena
Longitud (Cadena)	-> entero
Buscar (Cadena, Carácter)	-> entero
Concatenar (Cadena1, Cadena2)	-> Cadena
Extraer (Cadena, Posición, NumCaracteres)	-> Cadena

- 17.4. Diseñar el TAD Bolsa como una colección de elementos no ordenados y que pueden estar repetidos. Las operaciones del tipo abstracto: CrearBolsa, Añadir un elemento, BolsaVacia (verifica si tiene elementos), Dentro (verifica si un elemento pertenece a la bolsa), Cuantos (determina el número de veces que se encuentra un elemento), Union y Total. Realizar la especificación informal y formal considerando como constructores las operaciones CrearBolsa y Añadir.

ESPECIFICACIÓN INFORMAL

TAD Bolsa (Colección de elementos no ordenados que pueden estar repetidos).

Operaciones*CrearBolsa*

Crea una bolsa vacía.

Añadir (Bolsa, elemento)

Añade un elemento a la bolsa.

BolsaVacia (Bolsa)

Verifica que la bolsa no tiene elementos.

Dentro (elemento, Bolsa)

Verifica si un elemento pertenece a la bolsa

Cuantos (elemento, Bolsa)

Determina el número de veces que se encuentra un elemento en una bolsa

Union (Bolsa1, Bolsa2)

Devuelve una bolsa con los elementos de los dos argumentos.

Total (Bolsa)

Devuelve el número de elementos de una bolsa.

ESPECIFICACIÓN FORMAL

TAD Cadena (Secuencia de caracteres ASCII terminada por un byte nulo).

Sintaxis

*CrearBolsa	-> Bolsa
*Añadir (Bolsa, elemento)	-> Bolsa
BolsaVacia (Bolsa)	-> boolean
Dentro (elemento, Bolsa)	-> boolean
Cuantos (elemento, Bolsa)	-> entero
Union (Bolsa1, Bolsa2)	-> Bolsa
Total (Bolsa)	-> Bolsa

- 17.5. Diseñar el TAD Complejo para representar a los números complejos. Las operaciones que se deben definir: AsignaReal (asigna un valor a la parte real), AsignaImaginaria (asigna un valor a la parte imaginaria), ParteReal (devuelve la parte real de un complejo), ParteImaginaria (devuelve la parte imaginaria de un complejo), Modulo de un complejo y Suma de dos números complejos. Realizar la especificación informal y formal considerando como constructores las operaciones que deseé.

ESPECIFICACIÓN INFORMAL

TAD Complejo (Par de números reales que representan la parte real e imaginaria de un número complejo según el concepto matemático).

Operaciones

AsignaReal (Complejo, real).

Asigna un valor a la parte real de un número complejo.

AsignaImaginaria (Complejo, real).

Asigna un valor a la parte imaginaria de un número complejo.

ParteReal (Complejo).

Devuelve la parte real de un número complejo.

ParteImaginaria (Complejo).

Devuelve la parte imaginaria de un número complejo.

Modulo (Complejo).

Devuelve el módulo de un número complejo.

Suma (Complejo1, Complejo2).

Devuelve la suma de dos números complejos

ESPECIFICACIÓN FORMAL

TAD Complejo (Par de números reales que representan la parte real e imaginaria de un número complejo según el concepto matemático).

Sintaxis

*AsignaReal (Complejo, real)	-> Complejo
*AsignaImaginaria (Complejo, real)	-> Complejo
ParteReal (Complejo)	-> real
ParteImaginaria (Complejo)	-> real
Modulo (Complejo)	-> real
Suma (Complejo1, Complejo2)	-> Complejo

- 17.6. Diseñar el tipo abstracto de datos Vector con la finalidad de representar una secuencia de n elementos del mismo tipo. Las operaciones a definir: CrearVector (crea un vector n posiciones vacías), Asignar (asigna un elemento en la posición j), ObtenerElemento (devuelve el elemento que se encuentra en la posición j), SubVector (devuelve el vector comprendido entre las posiciones i, j). Realizar la especificación informal y formal considerando como constructores las operaciones que deseé.

ESPECIFICACIÓN INFORMAL

TAD Vector (secuencia de n elementos del mismo tipo).

Operaciones

CrearVector (entero).

Crea un vector n posiciones vacías.

Asignar (Vector, posición, elemento).

Asigna un elemento en la posición indicada en el segundo parámetro.

ObtenerElemento (Vector, posición).

Devuelve el elemento que se encuentra en la posición indicada en el segundo parámetro.

SubVector (Vector, inicial, final).

Devuelve el vector comprendido entre las posiciones indicadas en los parámetros finales.

ESPECIFICACIÓN FORMAL

TAD Vector (secuencia de n elementos del mismo tipo).

Sintaxis

*CrearVector (entero)	-> Vector.
*Asignar (Vector, entero, elemento)	-> Vector.
ObtenerElemento (Vector, entero)	-> elemento.
SubVector (Vector, entero, entero)	-> Vector.

- 17.7.** Diseñar el tipo abstracto de datos Matriz con la finalidad de representar matrices matemáticas. Las operaciones a definir: CrearMatriz (crea una matriz, sin elementos, de m filas por n columnas), Asignar (asigna un elemento en la fila i columna j), ObtenerElemento (obtiene el elemento de la fila i y columna j), Sumar (realiza la suma de dos matrices cuando tienen las mismas dimensiones), ProductoEscalar (obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor). Realizar la especificación informal y formal considerando como constructores las operaciones que deseé.

ESPECIFICACIÓN INFORMAL

TAD Matriz (Secuencia de elementos organizados en filas y columnas).

Operaciones

CrearMatriz (filas, columnas).

Crea una matriz, sin elementos, de las dimensiones que indican los argumentos.

Asignar (Matriz, fila, columna, elemento).

Asigna un elemento en la posición que indican los argumentos finales.

ObtenerElemento (Matriz, fila, columna).

Obtiene el elemento de la posición que indican los argumentos finales.

Sumar (Matriz1, Matriz2).

Realiza la suma de dos matrices cuando tienen las mismas dimensiones

ProductoEscalar (Matriz, valor).

Obtiene la matriz resultante de multiplicar cada elemento de la matriz por un valor.

ESPECIFICACIÓN FORMAL

TAD Matriz (Secuencia de elementos organizados en filas y columnas).

Sintaxis

*CrearMatriz (entero, entero)	-> Matriz.
Asignar (Matriz, entero, entero, elemento)	-> Matriz.
ObtenerElemento (Matriz, entero, entero)	-> Matriz.
Sumar (Matriz, Matriz)	-> Matriz.
ProductoEscalar (Matriz, valor)	-> Matriz.

- 17.8.** Implementar el TAD Conjunto con las operaciones especificadas en los ejercicios 17.1 y 17.2.

Análisis del problema

La implementación del tipo abstracto de datos debe incluir dos partes diferenciadas:

- representación de los datos.
- implementación de las operaciones descritas en la especificación.

Los archivos de *inclusión* o de **cabecera** se utilizan para agrupar en ellos variables externas, declaraciones de datos comunes y prototipos de funciones. Estos archivos de cabecera se incluyen en los archivos que contienen la codificación de las funciones, archivos fuente, y también en los archivos de código que hagan referencia a algún elemento del *archivo de inclusión*, con la directiva del preprocesador `#include`. Al implementar un TAD en C, se agrupa, en cierto modo se *encierra*, en estos archivos la representación de los datos y el interfaz del TAD, a su vez, representado por los prototipos de las funciones. De esta forma en los archivos de código fuente que utilicen el TAD hay que escribir la directiva

```
#include "tipodedato.h"
```

Para hacer la implementación lo más flexible posible, no se establece que el conjunto pueda tener un máximo de elementos. Esta característica exige el uso de asignación dinámica de memoria.

En el archivo de cabecera, `conjunto.h`, se realiza la declaración de la estructura que va a representar a los datos. El tipo de los datos puede ser cualquiera, entonces es necesario que `TipoDato` esté especificado antes de incluir `conjunto.h`. La constante `M`, que arbitrariamente toma el valor de 10, es el número de “huecos” o posiciones de memoria, que se reservan cada vez que hay que ampliar el tamaño de la estructura.

Archivo `conjunto.h`

```
#define M 10
typedef struct
{
    TipoDato* cto;
    int cardinal;
    int capacidad;
} Conjunto;

void conjuntoVacio (Conjunto* c);
int esVacio (Conjunto c);
void añadir (Conjunto* c, TipoDato elemento);
void retirar (Conjunto* c, TipoDato elemento);
int pertenece (Conjunto c, TipoDato elemento);
int cardinal (Conjunto c);
Conjunto unionC (Conjunto c1, Conjunto c2);
Conjunto interseccionC (Conjunto c1, Conjunto c2);
int incluido (Conjunto c1, Conjunto c2);
```

Este archivo de cabecera, `conjunto.h`, hay que incluirlo en todos los archivos con código C que vaya a utilizar el tipo `Conjunto`. Es importante recordar que antes de escribir la sentencia `include` hay que asociar un tipo predefinido a `TipoDato`. Por ejemplo, si los elementos del conjunto son las coordenadas de un punto en el plano:

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

typedef Punto TipoDato;
#include "conjunto.h"
```

Las funciones cuyos prototipos han sido ya escritos, se codifican y se guardan en el archivo `conjunto.c`. La compilación de `conjunto.c` da lugar al archivo con el código objeto que se ensamblará con el código objeto de otros archivos fuente que hacen uso del TAD `Conjunto`.

*Codificación***Archivo** conjunto.c

```
typedef struct {var(s)} Tipo;

typedef Tipo TipoDato;
#include "conjunto.h"

/* iguales() devuelve 1(cierto) si todos los campos lo son.
   La implementación depende del tipo concreto de los dato
   del conjunto.
*/
int iguales (TipoDato e1, TipoDato e2)
{
    return (e1.v1 == e2.v1) && (e1.v2 == e2.v2) ... ;
}

void conjuntoVacio(Conjunto* c)
{
    c -> cardinal = 0;
    c -> capacidad = M;
    c -> cto = (TipoDato*)malloc (M*sizeof(TipoDato));
}

int esVacio(Conjunto c)
{
    return (c.cardinal == 0);
}

void añadir (Conjunto* c, TipoDato elemento)
{

    if (!pertenece(*c, elemento))
    {
        /* verifica si hay posiciones libres,
           en caso contrario amplia el conjunto */
        if (c -> cardinal == c -> capacidad )
        {
            Conjunto nuevo;
            int k, capacidad;
            capacidad = (c -> capacidad + M)*sizeof(TipoDato)
            nuevo.cto = (TipoDato*) malloc(capacidad);

            for (k = 0; k < c -> capacidad; k++)
                nuevo.cto[k] = c -> cto[k];

            free(c -> cto);
            c -> cto = nuevo.cto;
        }
        c -> cto[c -> cardinal++] = elemento;
    }
}
```

```
void retirar (Conjunto* c, TipoDatos elemento)
{
    int k;

    if (pertenece (*c, elemento))
    {
        k = 0;
        while (!iguales (c -> cto[k], elemento)) k++;

        /* desde el elemento k hasta la última posición
           mueve los elementos una posición a la izquierda */

        for (; k < c -> cardinal ; k++)
            c -> cto[k] = c -> cto[k+1];

        c -> cardinal--;
    }
}

int pertenece (Conjunto c, TipoDatos elemento)
{
    int k, encontrado;
    k = encontrado = 0;

    while (k < c.cardinal && !encontrado)
    {
        encontrado = iguales (c.cto[k], elemento);
        k++;
    }
    return encontrado;
}

int cardinal(Conjunto c)
{
    return c.cardinal;
}

Conjunto unionC(Conjunto c1, Conjunto c2)
{
    Conjunto u;
    int k;
    u.cardinal = 0;
    u.capacidad = c1.capacidad;
    u.cto = (TipoDatos*)malloc (u.capacidad*sizeof(TipoDatos));

    for (k = 0; k < c1.capacidad; k++)
        u.cto[k] = c1.cto[k];
    u.cardinal = c1.cardinal;

    for (k = 0; k < c2.capacidad; k++)
        añadir (&u, c2.cto[k]);
    return u;
}
```

```

Conjunto interseccionC (Conjunto c1, Conjunto c2)
{
    Conjunto ic;
    int k, l;

    ic.cardinal = 0;
    ic.capacidad = c1.capacidad;
    ic.cto = (TipoDato*)malloc (u.capacidad*sizeof(TipoDato));

    for (k = 0; k < c1.capacidad; k++)
        for (l = 0; l < c1.capacidad; l++)
            if (iguales (c1.cto[k], c2.cto[l]))
            {
                annadir (&ic, c1.cto[k]);
                ic.cardinal++;
            }
    return ic;
}

int incluido (Conjunto c1, Conjunto c2)
{
    int k;

    if (c1.cardinal==0) return 1;
    for (k = 0; k < c1.capacidad; k++)
        if (!pertenece (c2, c1.cto[k]))
            return 0;

    return 1;
}

```

- 17.9.** Implementar el TAD *Bolsa* descrito en el ejercicio 17.4. Probar la implementación con un programa que invoque a las operaciones del tipo abstracto *Bolsa*.

Codificación (Consultar la página web del libro)

- 17.10.** Implementar el TAD *Cadena* descrito en el ejercicio 17.3. Probar la implementación con un programa que realice diversas operaciones con cadenas.

Análisis del problema

En primer lugar es conveniente definir el contenido de un fichero de cabecera para definir los prototipos de las funciones que se implementarán y el tipo de datos “cadena”.

Codificación

cadena.h

```

#define N 100
typedef char cadena [N];

cadena cadenavacia ();

```

```

cadena asignar (cadena cad1, cadena cad2);
int longitud (cadena cad);
int buscar (cadena cad, char c);
cadena concatenar (cadena cad1, cadena cad2);
cadena extraer (cadena cad, int pos, int NumCar);

```

Las funciones cuyos prototipos han sido ya escritos, se codifican en el archivo cadena.c.

```

#include cadena.h

cadena cadenavacia ()
{
    char *vacia = (char*) malloc(80);
    return vacia;
}

cadena asignar (cadena cad1, cadena cad2)
{
    return strcat (cad1, cad2);
}

int longitud (cadena cad)
{
    return (strlen (cad));
}

int buscar (cadena cad, char c)
{
    return (strchr (cad, c));
}

cadena concatenar (cadena cad1, cadena cad2)
{
    return (strcat(cad1,cad2));
}

cadena extraer (cadena cad, int pos, int NumCar);
{
    char *aux = (char*) malloc (numcar + 1);
    int i,j = 0;

    for (i = pos; i < pos+numcar; i++)
        aux[j] = cad[i];
    return aux;
}

```

Un programa que pruebe las funciones anteriores podría ser el siguiente:

```

main()
{
    cadena cad1;
    cadena cad2, cad3;

```

```

cad1 = cadenavacia ();
cad2 = cadenavacia ();
cad2 = cadenavacia ();

printf ("Introduzca una cadena de caracteres: ");
scanf ("%s", cad1);
printf ("\nSu cadena tiene %d caracteres.\n", longitud(cad1));

printf ("Esta es la cadena con las dos mitades intercambiadas\n");

mitad = (int)(longitud(cad1)/2);
cad2 = extraer (cad1, 0, mitad);
cad3 = extraer (cad1, mitad+1, mitad);
cad3 = concatenar (cad3, cad2);

puts (cad3);
}

```

Deberá ser compilado junto con el fichero `cadena.c` que contiene las implementaciones de las funciones.

- 17.11.** Implementar el TAD Matriz especificado en el ejercicio 17.7 con una estructura dinámica. Escribir un programa que haciendo uso del TAD Matriz se realicen operaciones diversas y escriba las matrices generadas.

Codificación (Consultar la página web del libro)

PROBLEMAS PROPUESTOS

- 17.1.** Implementar el TAD Complejo especificado en el ejercicio 17.5. Escribir un programa en el que se realicen diversas operaciones con números complejos.
- 17.2.** Implementar el TAD Vector especificado en el ejercicio 17.6 con una estructura dinámica
- 17.3.** Diseñar el TAD Grafo como un conjunto de nodos y de aristas. Las operaciones del tipo abstracto serán: *CrearGrafo*, *AñadirNodo*, *AñadirArista*, *GrafoVacio* (verifica si tiene nodos o aristas), *Recorrido en Profundidad*, *RecorridoenAnchura*, *Cuantos* (determina el número de nodos), *Union* de dos grafos y *Conectados* (Verifica si existe un camino entre dos nodos). Realizar la especificación informal y formal considerando como constructores las operaciones *CrearGrafo*, *AñadirNodo* y *AñadirArista*.
- 17.4.** Implementar el TAD Grafo especificado en el ejercicio anterior con una estructura dinámica. Escribir un progra
- grama que haciendo uso del TAD Grafo se realicen operaciones diversas.
- 17.5.** Diseñar el TAD Árbol Binario ordenado. Las operaciones del tipo abstracto serán: *CrearArbol*, *AñadirNodo*, *ArbolVacio* (verifica si tiene nodos), *Recorrido en Profundidad*, *RecorridoenAnchura*, *Cuantos* (determina el número de nodos), *Union* de dos árboles y *Equilibrar* el árbol. Realizar la especificación informal y formal considerando como constructores las operaciones *CrearArbol* y *AñadirNodo*.
- 17.6.** Implementar el TAD Árbol Binario ordenado especificado en el ejercicio anterior con una estructura dinámica. Escribir un programa que haciendo uso del TAD Árbol Binario ordenado se realicen operaciones diversas.
- 17.7.** Diseñar el TAD Buzón de Mensajes. Las operaciones del tipo abstracto serán: *CrearBuzon*, *AbrirBuzon*, *BuzonVacio*, *RecibirMensaje* (recibir el último mensaje),

EnviarMensaje, VaciarBuzon, DestruirBuzon. Realizar la especificación informal y formal considerando como constructor la operación *CrearBuzon*.

- 17.8.** Implementar el TAD Buzón de Mensajes especificado en el ejercicio anterior con una estructura dinámica. Escribir un programa que haciendo uso del TAD Buzón de Mensajes se realicen operaciones diversas.
- 17.9.** Diseñar el TAD Semáforo. Las operaciones del tipo abstracto serán: *CrearSemaforo* (Crear un semáforo y

ponerlo en un estado determinado), *DestruirSemaforo, AbrirSemaforo, CerrarSemaforo, EsperarSemaforo* (Esperar a que se abrir el semáforo y cerrarlo). Realizar la especificación informal y formal.

- 17.10.** Implementar el TAD Semáforo especificado en el ejercicio anterior. Escribir un programa que haciendo uso del TAD Semáforo se realicen operaciones diversas.