Tecnología de la Programación

Programación Modular y Abstracción de Datos

2020

Juan Antonio Sánchez Laguna

Grado en Ingeniería Informática

Facultad de Informática

Universidad de Murcia

TABLA DE CONTENIDOS

Programación modular

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible, aumentar la reusabilidad del código y facilitar su mantenimiento y depuración. Esto es especialmente útil y necesario al desarrollar aplicaciones grandes.

Un módulo es un conjunto de recursos, es decir, definiciones y declaraciones de variables, tipos y funciones que, normalmente, tienen cierta relación entre sí. Un módulo puede poner algunos de sus recursos a disposición de otros módulos. A este subconjunto de recursos se le denomina interfaz público del módulo.

Al hablar de módulos es habitual pensar en términos de cliente y proveedor, entendiendo por cliente al módulo cuyo código necesita usar alguna función, tipo o variable proporcionada por otro módulo: el proveedor. En la práctica, el autor del código cliente y del código del módulo proveedor puede que sea la misma persona, pero es útil suponer lo contrario.

El interfaz público de un módulo debe ser conocido, sin embargo, la implementación debe permanecer oculta a la vista del cliente. Esto permite evitar que futuros cambios en la implementación de las funciones, o en la representación de los tipos de datos públicos, afecten al funcionamiento del código cliente.

En cualquier aplicación compuesta por varios módulos debe haber uno principal cuyo código será el primero que se ejecute. Este módulo será cliente de otros módulos, pero ningún otro dependerá de él. Al mismo tiempo, es posible que el código de algún módulo sea también cliente de otros. Estas relaciones de dependencia dan lugar a un grafo en el que no deberían existir ciclos.

El interfaz público de cualquier módulo debe estar bien documentado. La documentación debe incluir una descripción de la utilidad del módulo y la lista de recursos públicos que proporciona: funciones, tipos y variables. Además, para cada función se debe especificar sus precondiciones y su efecto. Las precondiciones son las condiciones que deben cumplir los parámetros para que una función pueda llevar a cabo su tarea correctamente. El efecto de una función es la descripción de la tarea o resultado que produce.

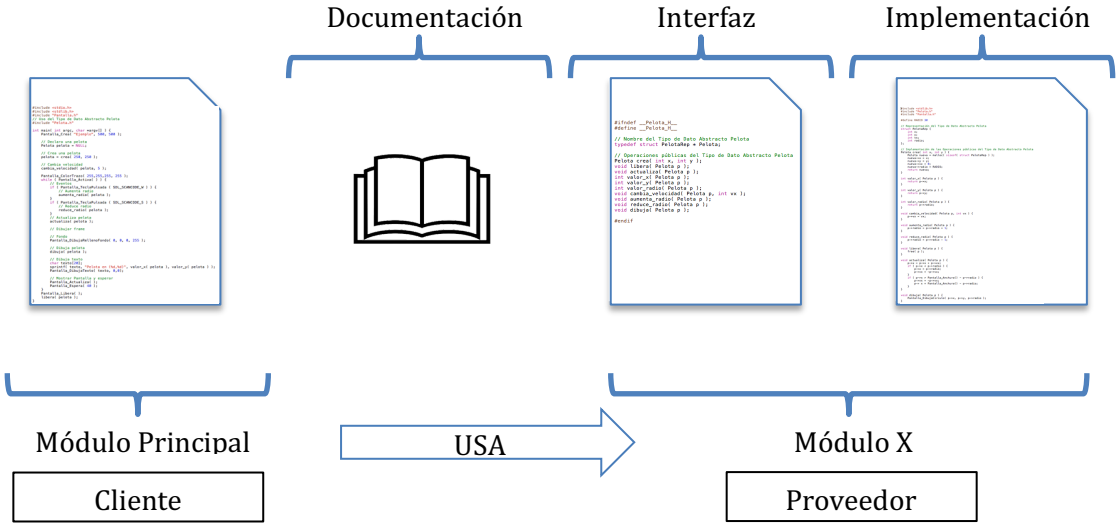

Fig. 1 Esquema básico de una aplicación modular

Creación de módulos en C

Para crear un módulo en C se escriben dos ficheros con el mismo nombre y distinta terminación. El primero, denominado fichero de cabecera, tendrá terminación `.h` y contendrá el interfaz público del módulo. Es decir, la declaración de todas las funciones, tipos, variables y macros que se desee ofrecer al resto de módulos. El segundo fichero tendrá terminación `.c` y contendrá el código de las funciones y las definiciones de los tipos de datos incluidas en el interfaz público y, si es necesario, otras que serán privadas, quedando su uso restringido al propio módulo.

El módulo principal de una aplicación escrita en C no suele proporcionar recursos públicos porque actúa como cliente de los demás módulos de la aplicación. Habitualmente consta de un único fichero llamado `main.c` que contiene la función principal y se coloca en una carpeta junto con el resto de módulos que forman la aplicación.

Para poder usar los recursos públicos de un módulo es necesario incluir su fichero de cabecera en el módulo donde se vayan a usar. Para incluir un fichero de cabecera se utiliza la directiva `#include` del preprocesador de C a la que se le indica la ruta (absoluta o relativa) y nombre del fichero a incluir.

Los módulos de la biblioteca estándar de C residen en carpetas conocidas del sistema operativo y no hace falta indicar su ruta completa, basta con indicar su nombre entre los símbolos < y >. Sin embargo, los ficheros cabecera desarrollados por el usuario pueden estar en cualquier carpeta. Para incluir este tipo de ficheros hay que indicar su ruta entre comillas dobles. Si sólo se indica su nombre se asume que están en la misma carpeta que el módulo donde esté la directiva `#include`.

Un mismo módulo puede ser usado múltiples veces dentro de una aplicación. Por tanto, su fichero de cabecera puede llegar a ser incluido varias veces. Para evitar que se produzcan errores de compilación debidos a declaraciones duplicadas, se deben proteger los ficheros de cabecera mediante directivas de compilación condicional (`#ifndef / #define / #endif`). La Fig. 2 muestra cómo hacerlo.

```
#ifndef __NombreModulo_H__
#define __NombreModulo_H__

// Definiciones y declaraciones públicas

#endif
```

Fig. 2 Plantilla para proteger un fichero cabecera de módulo usando compilación condicional

El bloque `#ifndef - #endif` encierra todas las definiciones y declaraciones del fichero cabecera. La directiva `#ifndef` comprueba si la etiqueta que le sigue está definida o no. Si no lo está se incluye todo lo que haya hasta la directiva `#endif`. Colocando como primera línea del bloque la definición de esa misma etiqueta, la próxima vez que se intente incluir este fichero la condición del `#ifndef` no se cumplirá porque la etiqueta ya estará definida. Por tanto, no se volverá a incluir el contenido del fichero. Evidentemente, la etiqueta debe ser única, por lo que ésta suele consistir en alguna variación del nombre del fichero.

Para crear una aplicación compuesta de varios módulos, se compilan todos y, posteriormente, se enlazan generando así un único fichero ejecutable. También es posible crear una biblioteca estática compilando uno o más módulos y generando un fichero (`.a` o `.lib`) que podrá ser usado para crear otras aplicaciones.

Precondiciones y gestión de errores en C

Las precondiciones son las condiciones que deben cumplir los parámetros para que una función pueda llevar a cabo su tarea correctamente y, en principio, no es necesario comprobarlas en el código fuente de la función. Sin embargo, es habitual incluir el código de comprobación para usarlo durante la fase de desarrollo. Al terminar esta fase se elimina para generar la aplicación o biblioteca definitiva.

Para comprobar que se cumplen las precondiciones se puede utilizar una macro llamada `assert` definida en el fichero de cabecera `assert.h` de la biblioteca estándar de C. Esta macro tiene un único parámetro que, de ser falso, detiene la ejecución del programa informando del fichero y línea de código donde se encuentre. Habitualmente se usa esta macro al inicio de la función para comprobar todas sus precondiciones.

Además, si al compilar el programa con `gcc` se usa la opción `–DNDEBUG`, o bien dentro del código se define la etiqueta `NDEBUG`, la macro `assert` no genera ningún código. Esto es útil para generar las versiones definitivas de las aplicaciones, cuando ya se ha superado la fase de pruebas.

Por otro lado, si durante la ejecución de una función se produce un error, hay que informar de lo ocurrido al cliente que la haya invocado. En C es habitual que las funciones devuelvan un código de error. Pero esto sólo es posible si entre los valores que puede devolver la función hay alguno que no se considere un resultado válido. Por ejemplo, al devolver la posición ocupada por un elemento en un array, el valor -1 sirve de código de error, pues no es un índice válido para un array en C.

Cuando esto no es posible, el módulo suele incluir una variable pública que almacenará el último código de error producido. En este caso, la documentación de las funciones en las que se puedan producir errores, deberá indicar a los usuarios del módulo la necesidad de comprobar el valor de dicha variable tras invocarlas.

En C, las variables definidas en un módulo son privadas al mismo. Para convertirlas en públicas se declaran e inicializan en el fichero `.c` del módulo y se vuelven a declarar, esta vez sin inicializar, en el fichero `.h`, pero anteponiendo a su declaración la palabra reservada `extern`.

Por último, para dar información detallada sobre la causa del error, se suele añadir una función pública que devuelva una cadena de texto con la descripción del último error producido, o la asociada al código de error pasado como parámetro.

Diseño de módulos

Al diseñar un módulo se debe procurar que éste tenga una tarea bien definida y que sea lo más independiente posible. Por ejemplo, en un módulo que tenga como misión facilitar el trabajo con fechas tiene poco sentido incluir una función que calcule el factorial de un entero. Del mismo modo, sería poco lógico que dicho módulo dependiera de otro cuyo propósito principal fuera facilitar el trabajo con vectores tridimensionales.

En general, un módulo debe ser lo más independiente posible y su interfaz debería ser lo más cohesionado posible. La independencia se logra reduciendo al mínimo el número de módulos de los que se depende. Por otro lado, se dice que el interfaz está cohesionado cuando sus funciones están muy relacionadas entre sí.

Ejemplo de módulo

A continuación, se muestra la documentación y código del módulo ficticio Pantalla.

Descripción del módulo

El módulo **Pantalla** ofrece varias funciones para mostrar gráficos en la pantalla. Para ello se apoya en el uso de la biblioteca SDL2

Recursos públicos del módulo

Tipos de datos

`Imagen`: Representa una imagen leída desde un fichero.

Funciones

```
void crea_pantalla( int ancho, int alto );
```

Este procedimiento crea una ventana lista para ser usada con el resto de funciones. El tamaño de la ventana será el indicado en los parámetros. Esta función debe ser ejecutada antes que cualquiera de las demás. Dentro de la ventana, el origen de coordenadas se encuentra en la esquina superior izquierda. Las coordenadas horizontales crecen hacia la derecha y van desde 0 hasta `ancho-1`. Las coordenadas verticales crecen hacia abajo y van desde 0 hasta `alto-1`.

```
void libera_pantalla();
```

Este procedimiento cierra la ventana creada con `crea_pantalla` y libera los recursos asociados a la misma.

```
void dibuja_linea( int x1, int y1, int x2, int y2 );
```

Este procedimiento dibuja una línea recta uniendo las coordenadas `(x1,y1)` y `(x2,y2)`.

```
void dibuja_circulo( int x, int y, int radio );
```

Este procedimiento dibuja un círculo con centro en las coordenadas `(x,y)` y el `radio` indicado.

```
Imagen lee_imagen( char * fichero );
```

Esta función lee el fichero cuya ruta y nombre se especifican en el parámetro y devuelve un valor de tipo `Imagen` que representa el contenido del mismo. El fichero debe contener una imagen guardada en formato BMP. Si el formato no es el correcto o se produjera algún error el valor devuelto sería `NULL`. La descripción del mensaje se puede obtener con la función `error_pantalla`.

```
void dibuja_imagen( Imagen img, int x, int y );
```

Este procedimiento dibuja la imagen `img` en la pantalla situando su esquina superior izquierda en las coordenadas `(x,y)`.

```
void libera_imagen( Imagen img );
```

Este procedimiento libera la memoria usada por la imagen `img`.

```
char * error_pantalla();
```

Este procedimiento devuelve una cadena de texto con la descripción del último error que se haya producido al ejecutar la función `lee_imagen`. No es necesario liberar la memoria asociada al mensaje.

Los ficheros del módulo descrito anteriormente serían `Pantalla.h` y `Pantalla.c` y su contenido sería similar al siguiente. Nótese que, al tratarse de un ejemplo ficticio, el fichero `Pantalla.c` no incluye el código real sino una simplificación.

```
#ifndef __Pantalla_H__
#define __Pantalla_H__

typedef struct ImagenRep * Imagen;

void crea_pantalla( int ancho, int alto );
void libera_pantalla();
void dibuja_linea( int x1, int y1, int x2, int y2 );
void dibuja_rectangulo( int x1, int y1, int ancho, int alto );
void dibuja_circulo( int x, int y, int radio );
Imagen lee_imagen( char * fichero );
void dibuja_imagen( Imagen img, int x, int y );
void libera_imagen( Imagen img );
char * error_pantalla();

#endif
```

Fig. 3 Fichero de cabecera del módulo ficticio Pantalla

El fichero `Pantalla.h` contiene la declaración del tipo `Imagen` y de las funciones públicas del módulo. Todo ello está protegido mediante directivas de compilación condicional del preprocesador de C para evitar duplicados al incluirlo desde otros módulos.

```
#include <SDL.h>
#include "Pantalla.h"

char * errores[] = { "El fichero no existe", "Error al leer el fichero",
                     "El formato es incorrecto"};

struct ImageRep {
  // Datos necesarios para representar la imagen (no mostrados)
};

void crea_pantalla( int ancho, int alto ) {
  // Código necesario para crear la ventana gráfica (no mostrado)
}

void libera_pantalla() {
  // Código necesario para liberar la ventana gráfica (no mostrado)
```

Fig. 4 Fichero de implementación del módulo ficticio Pantalla

El fichero `Pantalla.c` incluye la implementación del módulo. Es decir, la representación de los tipos y el código de las funciones. Aunque estas dos partes no se muestran completamente, sí aparecen dos aspectos importantes. En primer lugar, es habitual que el fichero `.c` incluya a su propio fichero cabecera, porque allí están todas las declaraciones necesarias. Además, se incluyen los ficheros cabecera de los que este módulo dependa. En este ejemplo sólo el de SDL que al estar instalada como una biblioteca del sistema se incluye usando <>.

En segundo lugar, la variable `errores` es privada al módulo y se usa para almacenar las cadenas de texto que serán devueltas por la función `Error`. Al ser cadenas definidas de esta forma no necesitan ser liberadas pues ocupan espacio en la zona de datos del programa, no en la memoria dinámica.

Abstracción de datos y Tipos de Datos Abstractos

La abstracción es un proceso mental por el cual se simplifica el objeto de estudio considerando sólo lo esencial y descartando lo demás. En programación se usa la abstracción para poder abordar problemas complejos.

Las funciones y procedimientos son un ejemplo de abstracción a nivel operacional. Permiten construir e incorporar al lenguaje nuevas operaciones que pueden utilizarse sin necesidad de conocer los detalles internos de su implementación. Una vez definidas, sólo se necesita saber qué hacen y cómo se usan.

A medida que el tamaño y la complejidad de los programas aumenta, las estructuras de datos que se utilizan en ellos también se vuelven más complejas. La abstracción de datos es un proceso que permite manejar esta complejidad a través del diseño, creación y uso de nuevos Tipos de Datos abstractos, es decir, Tipos de Datos que pueden usarse sin conocer los detalles de su implementación

Formalmente, un Tipo de Datos Abstracto (TDA) consiste en un conjunto de operaciones definidas sobre un dominio de valores cuyo comportamiento se especifica independientemente de la representación elegida para los valores.

La especificación, o parte pública de un TDA, es donde se define el nombre, el dominio de valores posibles y el conjunto de operaciones disponibles, al que también se le llama interfaz público. Por otro lado, en la implementación, o parte privada del TDA, se define la estructura de datos elegida para representar los valores del dominio y el código de las operaciones incluidas en su interfaz público.

Un mismo TDA se puede implementar de múltiples formas, pero para que una implementación se considere válida debe cumplir su especificación, garantizando que la representación de los valores del tipo sea privada (ocultación/privacidad de la representación) y que el acceso a la misma sólo sea posible a través de las operaciones incluidas en el interfaz público del TDA (protección de los datos).

En la práctica, un TDA se implementa mediante un módulo que proporciona tanto el nuevo Tipo de Datos como las funciones incluidas en su interfaz público. El módulo debe encapsular el TDA ocultando su representación y protegiendo los datos. De esta forma el código cliente del módulo puede usar el nuevo Tipo de Datos sin conocer los detalles de su implementación.

Por último, la separación entre especificación e implementación implícita en el concepto de TDA, es decir, la separación entre el qué hace y el cómo se hace, tiene muchos beneficios:

1. Facilita el desarrollo al cliente, pues éste no necesita conocer los detalles internos del TDA para utilizarlo en una aplicación, basta con que conozca su especificación, que es mucho más simple.
2. Aumenta la reusabilidad ya que un mismo TDA se puede usar en múltiples aplicaciones.
3. Simplifica el mantenimiento y la mejora de rendimiento, pues se puede modificar y mejorar la implementación del TDA sin alterar el código cliente porque la implementación del TDA seguirá respetando su especificación.
4. Aumenta la fiabilidad de las aplicaciones, pues es más fácil realizar pruebas independientes sobre los módulos que componen los distintos TDAs de un programa, lo que garantiza su corrección.

Especificación Informal de Tipos de Datos Abstractos

Una especificación informal es un documento donde aparece el nombre, el dominio de valores, el interfaz público del Tipo de Datos que define y una descripción de su propósito o utilidad, y del contexto para el que está diseñado. Es similar a un contrato en el que se indica qué hace el TDA, pero no dice nada sobre cómo lo hace.

Además, para cada operación del interfaz público del TDA se debe indicar su sintaxis y su semántica:

- La sintaxis incluye el nombre de la operación, la lista de parámetros que admite y el Tipo de Datos de los valores que devuelve.
- La semántica consiste en la descripción del efecto que produce y las precondiciones que deben satisfacer los parámetros para que la operación pueda realizar su función correctamente.

La especificación es la única información que se necesita conocer para poder utilizar el nuevo Tipo de Datos proporcionado por el TDA. El código cliente de un TDA no debe conocer ni tener acceso a la representación interna de dicho TDA. Debe limitarse a utilizarlo según se indique en su especificación.

Clasificación de TDAs

Se puede decir que un TDA es simple si sus valores representan elementos individuales (Punto, Número complejo, Persona), y diremos que un TDA es de tipo contenedor cuando sus valores representen colecciones de otros elementos (Conjunto, Lista, Bolsa). Por otro lado, se puede hablar de TDAs mutables o inmutables en función de si incluyen operaciones de modificación o no. Es decir, si los valores creados pueden modificarse (mutables) o permanecen inalterables hasta su fin (inmutables).

Cuestiones de Diseño de TDAs

Al diseñar un TDA, además de seleccionar la estructura de datos más apropiada para representar los valores del Tipo de Datos, se debe elegir el conjunto de operaciones que constituirá su interfaz público. Este conjunto debe incluir, al menos, una operación de creación. Si se utiliza memoria dinámica y su liberación debe hacerse de forma explícita, como en el caso de C, también será necesario incluir alguna operación de liberación. También puede ser útil incluir operaciones que combinen dos o más valores del tipo creando uno nuevo.

Además, es habitual y conveniente incluir operaciones de acceso (también llamadas tipo get) cuya función será obtener el valor de las propiedades del elemento representado por el TDA o algún resultado derivado de las mismas. Pero siempre se debe evitar el devolver punteros a estructuras internas del TDA que permitan al usuario conocer y modificar la representación interna del mismo.

Si el TDA se quiere hacer mutable se incluirán operaciones de modificación (también llamadas tipo set) que permitan alterar las propiedades de los valores del nuevo Tipo de Datos.

Finalmente, en el caso de los TDA de tipo contenedor suele ser necesario incluir operaciones para añadir, eliminar, buscar y aplicar operaciones a todos o algunos de los elementos de la colección.

Implementación de TDAs en C

La implementación de un TDA es la traducción a código en un lenguaje de programación concreto del comportamiento descrito en la especificación. Es donde se incluyen todos los detalles sobre cómo se guardan en memoria los valores del tipo y cómo funcionan las operaciones que constituyen su interfaz público.

Para implementar un TDA, se debe elegir una estructura de datos que permita representar los valores del dominio del nuevo Tipo de Datos e implementar las operaciones de su interfaz cumpliendo la especificación.

La implementación de cada TDA consistirá en un módulo que pueda compilarse de forma independiente garantizando la ocultación/privacidad de la representación y la protección de los datos. Al compilar el módulo se generará un fichero objeto que, junto con el fichero cabecera, será lo único que se le dará el usuario del TDA.

El nombre del módulo aludirá al del TDA que implementa y su fichero de cabecera (`.h`) contendrá la parte pública, es decir, el nombre del Tipo de Datos y las declaraciones de las funciones de su interfaz público. El fichero `.c` del módulo contendrá la parte privada, es decir, la definición de la estructura de datos elegida para representar los valores del Tipo de Datos y el código de las operaciones.

El nombre del TDA se definirá usando la estrategia del puntero opaco para garantizar la privacidad y protección de los datos. Es decir, se definirá como un puntero a una estructura incompleta (aún no definida) que se llamará como el Tipo de Datos, pero con el sufijo `Rep`. Dicha estructura será la que contenga todo lo necesario para representar los valores del Tipo de Datos, pero su definición no aparecerá en el fichero cabecera sino en el fichero `.c` del módulo.

```
typedef struct ImagenRep * Imagen;
```

Fig. 5 Definición del nombre del TDA Imagen mediante la estrategia del puntero opaco

Por ejemplo, si se está definiendo el TDA Imagen, en el fichero cabecera aparecería la definición de la Fig. 5. En el mismo fichero cabecera se declararán todas las funciones incluidas en el interfaz público del TDA, pero sin incluir su código fuente.

En el fichero .c del módulo que implemente el TDA se definirá la estructura de datos elegida para representar los valores del nuevo Tipo de Datos. Además, se incluirá el código de las funciones que compongan el interfaz público del TDA, y el de otras funciones y estructuras de datos auxiliares que se requieran para organizar correctamente el código del módulo. Estas últimas serán privadas quedando su uso restringido al propio módulo.

La estrategia del puntero opaco garantiza que el código cliente no tenga acceso a los campos de la estructura que representa el TDA garantizando así la protección de los datos. De hecho, si se intenta acceder a alguno de ellos se obtiene el error "*deference pointer to incomplete type*". Pero esta estrategia requiere el uso de memoria dinámica. Por tanto, todas las funciones que creen nuevos valores del Tipo de Datos usarán `malloc` y deberá existir, al menos, una función que sirva para liberar la memoria reservada.

El usuario del TDA sólo necesitará el fichero `.h` y el fichero objeto, por lo que, en principio, no sabrá cómo está representado, no conocerá la implementación de las funciones públicas ni la existencia de las privadas.

Ejemplo de TDA

En un programa se necesita trabajar con puntos en el plano. Concretamente, crear puntos y calcular distancias entre ellos. Las coordenadas serán de tipo entero y, una vez creado, cualquier punto podrá moverse a otra posición. Dado que en C no hay un tipo de datos específico para representar puntos en el plano, se creará un TDA Punto para usarlo en el programa. Una posible especificación informal del TDA Punto sería la siguiente:

Nombre: Punto

Descripción: Representa puntos en el plano como parejas de coordenadas.

Dominio de valores: Las coordenadas son parejas de valores de tipo entero.

Interfaz público:

- `Punto crea_punto( int x, int y )`
  Crea un punto en las coordenadas (`x, y`)
- `void libera_punto( Punto p )`
  Libera la memoria asociada al punto `p`
- `int get_x_punto( Punto p )`
  Recupera la coordenada horizontal del punto `p`
- `void get_y_punto( Punto p )`
  Recupera la coordenada vertical del punto `p`
- `void set_x_punto( Punto p, int x )`
  Modifica la coordenada horizontal del punto `p`
- `void set_y_punto( Punto p, int y )`
  Modifica la coordenada vertical del punto `p`
- `double distancia( Punto a, Punto b )`
  Calcula la distancia euclídea entre dos puntos

Este interfaz público consta de funciones de creación, acceso y modificación.

La función `crea_punto` es la única que permite crear un nuevo valor del Tipo de Datos. Lo hace a partir de un par de enteros que pasarán a ser las coordenadas del nuevo punto. Además, debido a las características propias de C, se ha incluido la función `libera_punto` que libera la memoria asociada a los elementos de tipo `Punto` cuando dejan de ser útiles en el programa.

Las funciones `get_x_punto` y `get_y_punto` son funciones típicas de acceso a las propiedades del Tipo de Datos: funciones "tipo get". Gracias a ellas será posible, por ejemplo, determinar si la posición del ratón está suficientemente cerca de un punto para seleccionarlo y moverlo a otro sitio.

Las funciones `set_x_punto` y `set_y_punto` son funciones típicas de modificación de propiedades del Tipo de Datos: funciones "tipo set". Incluir estas operaciones es lo que hace que el TDA se considere mutable pues, aunque los puntos se creen con unas coordenadas concretas, será posible cambiarlas en cualquier momento.

Por último, la función `distancia` permite calcular distancias entre puntos para poder cumplir con lo indicado en la descripción del ejemplo.

Para implementar el TDA Punto se escribe un módulo compuesto por dos ficheros: `Punto.h` y `Punto.c`. El primero aparece en la figura Fig. 6 y contiene toda la parte pública del TDA. El segundo aparece en la figura Fig. 7 y contiene la representación del TDA y el código de las operaciones que componen su interfaz público.

El fichero `Punto.h` contiene, en primer lugar, la declaración del nombre del TDA y después, el interfaz público del mismo. Como puede verse, aplicando la estrategia del puntero opaco, el tipo `Punto` se ha declarado como un puntero a una estructura que, en ese lugar del código, aún no está definida, es decir, como un puntero a una estructura incompleta. La definición de la estructura `PuntoRep` aparece sólo en el fichero `Punto.c` garantizando así la privacidad y protección de los datos del TDA.

A continuación, aparecen las declaraciones de todas las operaciones públicas del TDA. Excepto en el caso de `distancia`, el nombre de las funciones incluye el sufijo `_punto`. Habitualmente, en un mismo programa coexisten varios TDAs y todos suelen tener operaciones de creación, liberación, acceso y modificación. C no permite identificadores duplicados, así que para evitar duplicidades y resolver ambigüedades se suele añadir a los nombres de las operaciones alguna parte que identifique a qué TDA pertenecen. En algunos casos, como en el de la función `distancia`, puede evitarse si se considera que no va a haber ninguna otra similar. Pero también se podría haber denominado `distancia_entre_puntos`.

La documentación del TDA se puede generar automáticamente. Doxygen es una de las herramientas que facilita este proceso. Esta herramienta lee los comentarios situados delante de la función a documentar e interpreta algunas palabras clave como `\brief`, `\pre`, `\param` o `\return` generando documentos HTML, PDF y otros formateando el resultado para mostrar adecuadamente el efecto, precondiciones, parámetros o valor devuelto de las funciones comentadas.

En el fichero `Punto.h` se puede ver un ejemplo de esto. Además de la sintaxis de cada función, en el comentario que la precede, se ha descrito su semántica, es decir, su efecto y, de haberlas, las precondiciones de la operación usando la notación de Doxygen.

La Fig. 7 muestra el fichero `Punto.c` del módulo que implementa el TDA Punto. Al principio del fichero se incluyen, además del fichero cabecera del módulo, los de las bibliotecas que se necesita usar. En concreto `<stdlib.h>` para trabajar con memoria dinámica y `<math.h>` para hacer los cálculos matemáticos necesarios para obtener la distancia euclídea entre dos puntos.

En este fichero sí aparece la definición de la estructura `struct PuntoRep`. Para una misma especificación se pueden hacer múltiples implementaciones, y en este caso se ha decidido representar el punto con dos valores de tipo entero. Pero también se podría haber hecho con un array con capacidad para dos enteros.

La función `crea_punto` es fundamental y refleja claramente el esquema a seguir en cualquier función de creación de TDAs. En su código se declara una variable de tipo `Punto`, y a esta variable se le asigna la dirección de memoria devuelta tras reservar suficiente espacio como para almacenar una estructura `struct PuntoRep`. Es decir, se ha creado un nuevo Punto en memoria dinámica. Después, tras dar valor a todos los campos que forman la estructura, se devuelve la dirección de esta. El código que usa el TDA guardará esta dirección en otra variable, también de tipo `Punto`, que será la que contenga el nuevo punto creado.

La función `distancia` simplemente calcula la distancia euclídea usando las coordenadas de los dos puntos que recibe como parámetro y las funciones `pow` y `sqrt` de la biblioteca math.h

```
#ifndef __Punto_H__
#define __Punto_H__

/**
  \brief TDA Punto
*/
typedef struct PuntoRep * Punto;

/**
  \brief Crea un punto en las coordenadas (x,y)
  \param x Coordenada horizontal del punto
  \param y Coordenada vertical del punto
*/
Punto crea_punto( int x, int y );

/**
  \brief Libera la memoria asociada al punto
  \param p Punto cuya memoria se desea liberar
*/
void libera_punto( Punto p );

/**
  \brief Recupera la coordenada horizontal del punto p
  \param p Punto del que recuperar su coordenada
  \return coordenada horizontal del punto p
*/
int get_x_punto( Punto p );

/**
  \brief Recupera la coordenada vertical del punto p
  \param p Punto del que recuperar su coordenada
  \return coordenada vertical del punto p
*/
int get_y_punto( Punto p );

/**
  \brief Modifica la coordenada horizontal del punto p
  \param p Punto al que modificar su coordenada
  \param x Nuevo valor para la coordenada horizontal
*/
void set_x_punto( Punto p, int x );

/**
  \brief Modifica la coordenada vertical del punto p
  \param p Punto al que modificar su coordenada
  \param y Nuevo valor para la coordenada vertical
*/
void set_y_punto( Punto p, int y );

/**
  \brief Calcula la distancia entre dos puntos
  \param a Primero de los puntos
  \param b Segundo de los puntos
  \return Distancia euclídea entre los puntos a y b
*/
double distancia( Punto a, Punto b );

#endif
```

Fig. 6 Fichero cabecera con el interfaz público del módulo que implementa el TDA Punto

```
#include "Punto.h"
#include <stdlib.h>
#include <math.h>

struct PuntoRep {
  int x;
  int y;
};

Punto crea_punto( int x, int y ) {
  Punto nuevo = malloc( sizeof( struct PuntoRep ) );
  nuevo->x = x;
  nuevo->y = y;
  return nuevo;
}

void libera_punto( Punto p ) {
  free( p );
}

int get_x_punto( Punto p ) {
  return p->x;
}

int get_y_punto( Punto p ) {
  return p->y;
}

void set_x_punto( Punto p, int x ) {
  p->x = x;
}

void set_y_punto( Punto p, int y ) {
  p->y = y;
}

double distancia( Punto a, Punto b ) {
  return sqrt( pow( b->x - a->x, 2 ) + pow( b->y - a->y, 2 ) );
}
```

Fig. 7 Fichero .c del módulo que implementa el TDA Punto

La Fig. 8 muestra un ejemplo de módulo principal en el que se hace uso del TDA Punto. Para poder usar el nuevo Tipo de Datos se debe incluir el fichero `Punto.h`. Hecho esto, es posible declarar variables de tipo `Punto`, y como se ve en el código de la función principal, son inicializadas usando la función `crea_punto`. Después, se modifican las coordenadas de una de ellas y se calcula la distancia entre los dos puntos. Finalmente, se libera la memoria asociada a los puntos creados.

También es posible crear nuevas funciones que usen el tipo Punto. Por ejemplo, la función `punto_medio` calcula las coordenadas del punto medio entre dos dados y las devuelve como un nuevo Punto.

```
#include <stdio.h>
#include "Punto.h"

Punto punto_medio( Punto a, Punto b ) {
  int xm = ( get_x_punto( a ) + get_x_punto( b ) ) / 2;
  int ym = ( get_y_punto( a ) + get_y_punto( b ) ) / 2;
  Punto medio = crea_punto( xm, ym );
  return medio;
}

int main() {
  Punto a = crea_punto( 1, 2 );
  Punto b = crea_punto( 2, 3 );
  set_y_punto( b, 2 );
  printf( "Distancia = %f\n", distancia( a, b ) );
  libera_punto( a );
  libera_punto( b );
}
```

Fig. 8 Ejemplo de uso del TDA Punto

Finalmente, como ejemplo avanzado, considérese el caso en el que en un segundo módulo se implementara un TDA para representar rectas en el plano. Este módulo bien podría ser cliente del TDA Punto, pues es habitual crear una recta a partir de dos puntos, tener que calcular la distancia entre un punto y una recta, o determinar el punto de corte entre dos rectas. En todos estos casos se usaría el tipo `Punto` como parámetro o como Tipo de Datos a devolver por las operaciones del TDA Recta.

El caso de la función que determina y devuelve el punto de corte entre dos rectas es muy interesante porque, en ocasiones, dos rectas no se cortan. En estos casos, y gracias al uso de la estrategia del puntero opaco, la función podría devolver el valor `NULL` para indicar la ausencia de punto de corte. Evidentemente, esto habría que especificarlo en la parte que describe el efecto de la función en su documentación como se muestra en la Fig. 9.

```
/**
  \brief Determina el punto de corte entre dos rectas
  \param a Primera recta
  \param b Segunda recta
  \return Punto de corte o NULL si las rectas son paralelas
*/
Punto punto_de_corte( Recta a, Recta b );
```

Fig. 9 Especificación informal de la función punto_de_corte

¿Por qué usar la estrategia del puntero opaco?

En algunas ocasiones se crean módulos que ofrecen un nuevo Tipo de Datos definido mediante alguna estructura de datos en su fichero cabecera. En dicho fichero de cabecera también aparece la declaración del conjunto de operaciones necesarias para crear, utilizar y combinar valores del nuevo Tipo de Datos. Y la implementación de las operaciones aparece en el fichero `.c` del módulo.

Sin embargo, estos módulos no implementan un TDA, pues el fichero cabecera se incluye en el código del cliente. Esto le da acceso a la representación del Tipo de Datos, y, por tanto, le permite acceder directamente a los campos que lo forman. Así es posible crear valores fuera del dominio del TDA, o incluso, sin sentido según la definición de este. También puede suceder que algunas de las operaciones del módulo fallen al no estar preparadas para tratar estos casos sin semántica. Además, si el código cliente llega a depender de aspectos concretos de la representación del Tipo de Datos, ciertas actualizaciones o mejoras realizadas en la implementación del mismo serían incompatibles con el código del cliente.

Por todo ello en este texto se recomienda usar punteros a estructuras incompletas para representar los TDA, es decir, un puntero a una estructura que en el momento de usarla para definir el nombre del TDA aún no está definida. Para usar esta estrategia[1] en C se define el nombre del Tipo de Datos en el fichero cabecera y la estructura que lo representa en el fichero con la implementación del TDA. De este modo, el módulo que implementa el TDA se puede compilar por separado, y al usuario del TDA sólo se le pasa el fichero cabecera y el resultado de la compilación.

Así pues, es posible crear programas en los que se use el TDA proporcionado, pero sin conocer la representación interna del Tipo de Datos, ni la implementación de sus operaciones. Es decir, se garantiza la privacidad. Además, desde fuera del módulo que implementa el TDA no es posible acceder a los campos de la estructura que lo representa, garantizando así la protección de los datos.

Esta estrategia también mejora la eficiencia del programa ya que todas las variables y parámetros del nuevo Tipo de Datos son en realidad punteros. Por lo tanto, independientemente de la cantidad de campos que tenga la estructura a la que apunte dicho puntero, el paso de parámetros y las asignaciones se resuelven copiando el valor de un único puntero.

Al mismo tiempo resulta sencillo escribir funciones que modifiquen los valores del tipo. Al recibir como parámetro un puntero a la estructura, en lugar de una copia de la misma, la opción de modificar los campos siempre es posible.

Además, gracias al uso que se hace de `typedef` asociando el nombre del Tipo de Datos al puntero a la estructura incompleta, resulta innecesario usar los operadores & y * a la hora de trabajar con los valores del Tipo de Datos desde el código cliente, lo que reduce la complejidad y aumenta la legibilidad del mismo.

Finalmente, el hecho de que cualquier valor del Tipo de Datos definido por el TDA sea un puntero permite usar el valor `NULL` para determinar si una variable contiene un valor correctamente creado o no, y también, como valor indicativo de error en funciones que devuelvan valores del Tipo de Datos.

---

[1] Bridge pattern: Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1995 (ISBN:0-201-63361-2)