Programación estructurada 05

En esta unidad aprenderás a:

- Comprender la justificación de la programación estructurada
- Conocer en qué consiste la programación estructurada
- Utilizar la la programación estructurada en el diseño de aplicaciones
- Mantener mejor las aplicaciones

5.1 La visión clásica de la programación estructurada: la programación sin *goto*

La visión clásica de la programación estructurada se refiere al control de ejecución. El control de su ejecución es una de las cuestiones más importantes que hay que tener en cuenta al construir un programa en un lenguaje de alto nivel. La regla general es que las instrucciones se ejecuten sucesivamente una tras otra, pero diversas partes del programa se ejecutan o no dependiendo de que se cumpla alguna condición. Además, hay instrucciones (los bucles) que deben ejecutarse varias veces, ya sea en número fijo o hasta que se cumpla una condición determinada.

Sin embargo, algunos lenguajes de programación más antiguos (como Fortran) se apoyaban en una sola instrucción para modificar la secuencia de ejecución de las instrucciones mediante una transferencia incondicional de su control (con la instrucción *goto*, del inglés "go to", que significa "ir a"). Pero estas transferencias arbitrarias del control de ejecución hacen los programas muy poco legibles y difíciles de comprender. A finales de los años sesenta, surgió una nueva forma de programar que reduce a la mínima expresión el uso de la instrucción *goto* y la sustituye por otras más comprensibles.

Esta forma de programar se basa en un famoso teorema, desarrollado por Edsger Dijkstra, que demuestra que todo programa puede escribirse utilizando únicamente las tres estructuras básicas de control siguientes:

- **Secuencia:** el bloque secuencial de instrucciones, instrucciones ejecutadas sucesivamente, una detrás de otra.

- **Selección:** la instrucción condicional con doble alternativa, de la forma "*if* condición *then* instrucción-1 *else* instrucción-2".

- **Iteración:** el bucle condicional "*while* condición *do* instrucción", que ejecuta la instrucción repetidamente mientras la condición se cumpla.

Los programas que utilizan sólo estas tres instrucciones de control básicas o sus variantes (como los bucles *for, repeat* o la instrucción condicional *switch-case*), pero no la instrucción *goto*, se llaman **estructurados.**

Ésta es la noción clásica de lo que se entiende por **programación estructurada** (llamada también **programación sin *goto***) que hasta la aparición de la programación orientada a objetos se convirtió en la forma de programar más extendida. Esta última se enfoca hacia la reducción de la cantidad de estructuras de control para reemplazarlas por otros elementos que hacen uso del concepto de polimorfismo. Aun así los programadores todavía utilizan las estructuras de control (*if, while, for,* etc.) para implementar sus algoritmos porque en muchos casos es la forma más natural de hacerlo.

Una característica importante en un programa estructurado es que puede ser leído en secuencia, desde el comienzo hasta el final sin perder la continuidad de la tarea que

cumple el programa, lo contrario de lo que ocurre con otros estilos de programación. Este hecho es importante debido a que es mucho más fácil comprender completamente el trabajo que realiza una función determinada si todas las instrucciones que influyen en su acción están físicamente contiguas y encerradas por un bloque. La facilidad de lectura, de comienzo a fin, es una consecuencia de utilizar solamente tres estructuras de control, y de eliminar la instrucción de transferencia de control *goto*.

5.2 La visión moderna de la programación estructurada: la segmentación

La realización de un programa sin seguir una técnica de programación produce frecuentemente un conjunto enorme de sentencias cuya ejecución es compleja de seguir, y de entender, pudiendo hacer casi imposible la depuración de errores y la introducción de mejoras. Se puede incluso llegar al caso de tener que abandonar el código preexistente porque resulte más fácil empezar de nuevo.

Cuando en la actualidad se habla de programación estructurada, nos solemos referir a la división de un programa en partes más manejables (usualmente denominadas **segmentos** o **módulos**). Una regla práctica para lograr este propósito es establecer que cada segmento del programa no exceda, en longitud, de una página de codificación, o sea, alrededor de 50 líneas.

Así, la visión moderna de un programa estructurado es un compuesto de segmentos, los cuales puedan estar constituidos por unas pocas instrucciones o por una página o más de código. Cada segmento tiene solamente una entrada y una salida, asumiendo que no poseen bucles infinitos y no tienen instrucciones que jamas se ejecuten. Encontramos la relación entre ambas visiones en el hecho de que los segmentos se combinan utilizando las tres estructuras básicas de control mencionadas anteriormente y, por tanto, el resultado es también un programa estructurado.

Cada una de estas partes englobará funciones y datos íntimamente relacionados semántica o funcionalmente. En una correcta partición del programa deberá resultar fácil e intuitivo comprender lo que debe hacer cada módulo.

En una segmentación bien realizada, la comunicación entre segmentos se lleva a cabo de una manera cuidadosamente controlada. Así, una correcta partición del problema producirá una nula o casi nula dependencia entre los módulos, pudiéndose entonces trabajar con cada uno de estos módulos de forma independiente. Este hecho garantizará que los cambios que se efectúen a una parte del programa, durante la programación original o su mantenimiento, no afecten al resto del programa que no ha sufrido cambios.

Esta técnica de programación conlleva las siguientes ventajas:

a) El coste de resolver varios subproblemas de forma aislada es con frecuencia menor que el de abordar el problema global.

b) Facilita el trabajo simultáneo en paralelo de distintos grupos de programadores.

c) Posibilita en mayor grado la reutilización del código (especialmente de alguno de los módulos) en futuras aplicaciones.

Aunque no puede fijarse de antemano el número y tamaño de estos módulos, debe intentarse un compromiso entre ambos factores. Si nos encontramos ante un módulo con un tamaño excesivo, podremos dividir éste a su vez en partes (nuevos módulos) más manejables, produciéndose la sucesiva división siempre desde problemas generales a problemas cada vez menos ambiciosos y, por tanto, de fácil desarrollo y seguimiento. Así, esta división toma la forma de un árbol cuya raíz es el programa principal que implementa la solución al problema que afrontamos utilizando uno o varios módulos que realizan partes de dicha solución por sí solos o invocando a su vez otros módulos que solucionan subproblemas más específicos. A esta aproximación se la denomina **diseño descendente** o ***top-down***, como queda esquematizado en la siguiente figura:

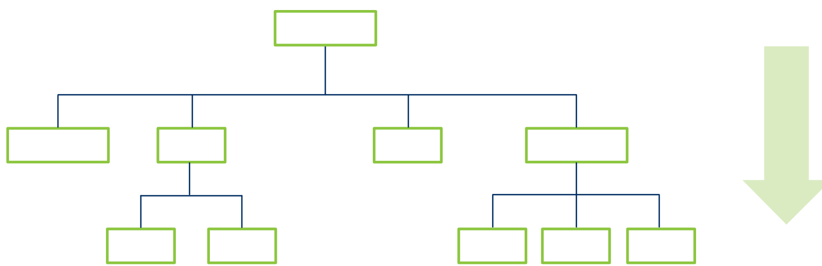


Fig. 5.1. *Diseño descendente.*

El carácter autocontenido de los módulos o librerías hace que pueda ocultarse el funcionamiento interno de las funciones contenidas en un módulo, ya que para utilizarlas basta con saber con qué nombre y argumentos se invocan y qué tipo de valores devuelven. Al reunirlas en un módulo, se realza la relación entre las mismas separándolas del resto del programa.

Esta ocultación de los detalles se denomina **encapsulación** y se alcanza dividiendo el código del programa en dos ficheros diferenciados: un fichero (con extensión ".h") que incluye la declaración de los tipos de datos y de las funciones gracias a lo cual se sabe cómo acceder y utilizar cada una de las mismas y otro (con extensión ".c") que contiene el código de cada una de las funciones declaradas en el .h.

Al compilar este último queda transformado en código objeto (al cual ya no se puede acceder para su lectura o modificación) y puede distribuirse conjuntamente con el fichero de declaración (el que acaba en .h), para su uso por parte de terceros sin riesgo alguno de alteración de la funcionalidad original (ésta quedó encapsulada u oculta).

Esto es así porque para hacer uso de las funciones incluidas en el módulo únicamente necesitaremos conocer la información que nos proporciona el fichero de declaración: el nombre, tipos de datos de los argumentos y valores de retorno de las funciones. No es necesario conocer los detalles de implementación (sentencias que se ejecutan dentro de una función).

Ejemplo práctico

1. Supongamos que queremos desarrollar una aplicación para que dos usuarios jueguen al ajedrez. En primer lugar, necesitamos una serie de funciones asociadas al tablero: una que coloque todas las piezas de ambos jugadores en las posiciones iniciales, otra que determine si se ha producido el final de la partida, otra que confirme la presencia de una pieza del jugador en juego en una determinada casilla, otra que confirme que el movimiento no se sale del tablero, o que ese tipo de pieza puede realizarlo, y una última que lleve a efecto el movimiento de una pieza del jugador en juego (eliminando del tablero a otra pieza del jugador contrario si procede). Para estas últimas puede ser preciso utilizar a su vez otro módulo que represente las funciones de movimiento asociadas a cada tipo de pieza.

La descomposición en módulos del programa quedaría:

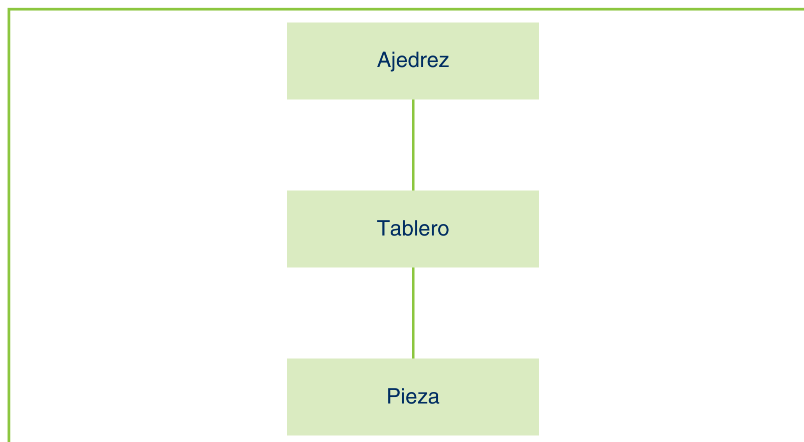


Fig. 5.2. *Descomposición del programa en módulos.*

Así, el contenido del fichero de libreríatablero.h sería:

```
enum columnas { a, b, c, d, e, f, g, h };
struct tablero {
      int posx;
      enum columnas posy;
      struct pieza mipieza;
      struct tablero *siguiente;
}

int Final(struct tablero *mitablero);
void Inicio(struct tablero *mitablero);
int Pertenece(struct tablero *mitablero, int posx, enum columnas posy);
void Dibuja(struct tablero *mitablero);
int FueraTablero(struct tablero *mitablero, int x1, enum columnas y1, int x2, enumcolumnas y2);
int Valido_T(struct tablero *mitablero, int x1, enum columnas y1, int x2, enum columnas y2);

void Mover(struct tablero *mitablero, int x1, enum columnas y1, int x2, enum columnas y2);
```

Así, el fichero de declaración pieza.h incluiría el siguiente código:

```
enum color { blanco, negro };
enum tipo { peon, torre, caballo, alfil, reina, rey }
struct pieza {
      enum color micolor;
      enum tipo mitipo;
}
int Valido_P(struct pieza *mipieza, int x1, int y1, int x2, int y2);
```

De esta forma, con el conocimiento público de ambos ficheros de declaración (tablero.h, pieza.h) podremos hacer uso de las funciones de los módulos *tablero* y *pieza* declaradas en estos ficheros, a pesar de nuestra ignorancia completa de cómo fueron implementadas. Lo ignoramos, ya que no tenemos acceso a los ficheros de implementación o código fuente (tablero.c, pieza.c); únicamente disponemos del código objeto (ablero.o, pieza.o) para su ejecución posterior.

Evidentemente, desde el código que haga uso de alguna de las citadas funciones de la librería, hemos de invocar los ficheros de declaración correspondientes. Para ello, utilizaremos la directiva de C *include,* tal y como hacíamos para utilizar las funciones de entrada/salida con stdio.h o las de cadenas de caracteres con strong.h, sólo que cambiaremos los símbolos < y > por dobles comillas ", por no tratarse de una librería estándar, sino creada por el propio programador y cuya ubicación, por tanto, no es la del directorio que alberga dichas librerías, en la cual el compilador busca por defecto, sino la especificada entre dobles comillas. En caso de no especificarse ninguna ruta dentro de las dobles comillas, sería buscada en el directorio actual. Por tanto, podemos indicar que esta técnica persigue obtener una descomposición en forma de árbol que permita el fraccionamiento del programa en módulos que resulten fáciles de entender, desarrollar, probar y mantener, y que agrupen las funciones y datos que estén estrechamente relacionadas entre sí.

Ejemplo práctico

2 El siguiente programa implementa apertura de ajedrez *Amar*:

```
#include "tablero.h"
main()
{
      struct tablero mipartida;
      Inicio(&mipartida);
      Mover(&mipartida, b, 1, a, 3);
      Dibuja(&mipartida);
}
```

Al igual que las funciones *Mover* y *Valido* de la librería o módulo *tablero* invocan a una función del módulo *pieza*, el código de la primera debe incluir el fichero de declaración de la segunda:

```
#include "pieza.h"
```

5.3 Programación estructurada y orientación a objetos

Debido a que una forma habitual de realizar la partición en módulos consiste en agrupar las funciones que permiten operar con un determinado conjunto (o estructura) de datos, existe una analogía evidente con la programación orientada a objetos. De hecho, ésta puede verse como la continuación lógica de aquélla. Así pues, realizaremos una comparación entre ambas técnicas de programación, resaltando las semejanzas existentes entre ellas.

La mente humana utiliza patrones o modelos que faciliten la comprensión del complejo mundo real. Así, estos patrones mentales de comportamiento abstraen aquellas características de un elemento o situación necesarias para su comprensión ignorando los detalles que resulten irrelevantes. Este proceso es conocido como abstracción y resulta esencial en el razonamiento humano, ya que gracias a él podemos manejar sistemas de gran complejidad.

La abstracción ha sido un aspecto clave de la programación desde sus comienzos. Al principio, los programadores enviaban instrucciones binarias a los ordenadores. Posteriormente, se empleó el lenguaje ensamblador, que no es más que el resultado de un proceso de abstracción del código máquina y que nos permite representar secuencias de bits por símbolos más inteligibles e intuitivos denominados nemotécnicos.

El siguiente nivel de abstracción se alcanzó cuando se crearon las instrucciones definidas por el usuario, las macroinstrucciones que abstraían a las propias instrucciones. En un proceso de mayor abstracción surgieron los lenguajes de alto nivel (como Fortran, C, Pascal, Cobol, etc.), que evitan al programador tener que pelearse con el tan denostado código máquina.

La **programación estructurada** y **la orientación a objetos** constituyen los pasos finales de este camino, ya que permiten a los programadores escribir código sin necesidad de conocer los detalles de la implementación de funciones de una librería y objetos respectivamente. Su implementación permanece oculta a los programadores, liberándoles de los condicionantes que esto imponía.

De ese modo, un programador no necesita saber cómo se representan internamente los datos de un tablero de ajedrez. Tan sólo necesitará saber cómo mover, iniciar una partida o validar un determinado movimiento.

Lo realmente complicado de ambos paradigmas de programación es encontrar los modelos o patrones conceptuales correctos para el problema y no la implementación de los mismos. Realmente, el diseño adquiere la máxima importancia que no tenía en la programación tradicional.

La propuesta de la moderna programación estructurada y de la orientación a objetos consiste principalmente en intentar superar la complejidad inherente a los problemas del mundo real gracias a la abstracción de nuestro conocimiento sobre tales problemas y su encapsulación dentro de módulos o librerías y objetos respectivamente. Se utiliza el término encapsular porque los objetos y las librerías pueden verse como "cáp-

sulas" a modo de barrera conceptual que separa un conjunto de valores y operaciones, que poseen un substrato conceptual idéntico, del resto del sistema. Así, en el ejemplo número 2 hemos encapsulado los datos del tablero (piezas y posiciones) y las operaciones (iniciar, mover, validar y dibujar) dentro de la misma barrera conceptual que ha surgido de la abstracción de la idea de tablero. Así, tanto objetos como librerías tienen dos partes bien diferenciadas: la **declaración** y la **implementación.** La declaración o interfaz es de dominio público, conocido por las demás partes del programa. En la implementación se explicitan y codifican los elementos necesarios para responder a lo especificado en la declaración. Esta separación permite que cada objeto definido en nuestro sistema pueda ocultar su implementación y hacer pública su interfaz de comunicación.

La ocultación de la información permite aislar una parte del programa del resto, y que el código de una parte sea revisado o extendido sin afectar a la integridad de las demás partes.

En este sentido, es interesante resaltar que la opción de encapsular datos y operaciones en librerías y objetos facilita enormemente que se puedan utilizar en una aplicación ajena. Se puede, por tanto, empezar a hablar de una relación consumidor/proveedor de código. La programación convencional daba más importancia a la relación entre el programador y su código, que a la relación entre el código y las aplicaciones en que se usa.

El proceso de construcción del software resultante de aplicar las dos propiedades ya comentadas anteriormente sería:

1. Gracias a la abstracción se agruparán funcionalidad e información en partes diferenciadas. El sistema estará constituido entonces por partes que se invocan entre sí.
2. A continuación, se definirá una interfaz pública que permite la utilización de cada parte del sistema, manteniendo oculta la implementación en el interior de ella.

Resumiendo las analogías existentes entre ambas técnicas de programación, se puede decir que la abstracción facilita la comprensión de la división de un programa en partes y la encapsulación permite aislar la implementación de cada una de las partes, aumentando la estabilidad y reutilización del código.

En las unidades siguientes, seguiremos esta idea de asociar la programación estructurada a la orientación a objetos relacionando cada módulo/librería con las operaciones a realizar sobre una estructura de datos.

En el ejemplo práctico 3 vamos a ver una aplicación de la programación estructurada a la gestión de los productos que pone a la venta un determinado comercio. Estos productos podrán ser tanto libros como películas en DVD.

Ejemplo práctico

3 En la aplicación contaremos con un módulo para realizar operaciones con artículos, como dar de alta y de baja un producto, visualizarlos en pantalla, vender un determinado número de ejemplares o reponer stock de ese producto. La estructura de los datos correspondientes a un producto de dicho comercio sería:

```
struct articulo
{
	char cod_art[10];
	float precio;
	unsigned int existencias;
};
```

También contaremos con estructuras que agrupen la información relativa a los libros por un lado y a las películas por otro, de forma que pueda consultarse el código de producto por el autor de un libro, el director de una película o por sus nombres, así como dar de alta y de baja los libros y películas. Por ejemplo, las estructuras *libro* y *DVD* podrían ser:

```
enum genero { historica, intriga, rosa, viaje, autoayuda, aventuras, ensayo };
struct libro
{
	char cod_art[10];
	char titulo[20];
	char autores[30];
	char editorial[10];
	enum genero migenero;
};
struct DVD
{
	char cod_art[10];
	char titulo[20];
	char director[10];
	char actores[30];
	int año;
};
```

Así, la única relación entre la estructura *articulo* y la descripción del mismo en la estructura *libro* o *DVD* es el mismo valor del campo *cod_art*. De esta forma, es el programa principal el que se encarga de relacionarlos. La estructura en forma de árbol de las librerías de nuestra aplicación quedaría de la siguiente forma:

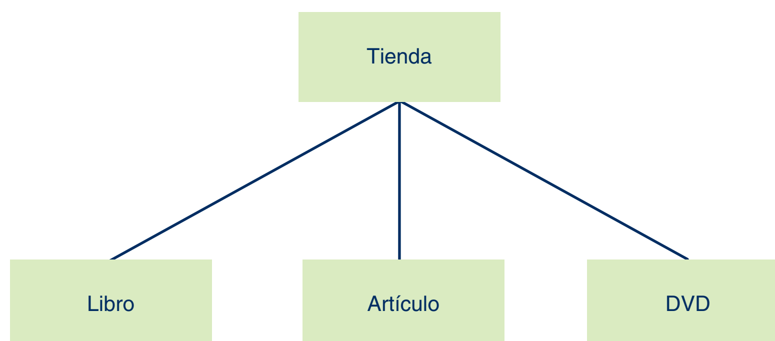


Fig. 5.3. *Estructura de árbol de la aplicación.*

Se podría haber optado por incluir un campo en la estructura de tipo *articulo* que se correspondiera con los datos del libro o DVD. Así descargamos al programa principal de la responsabilidad de relacionar ambas estructuras mediante el código del artículo y podríamos mostrar la información sobre el precio y stock junto con los detalles del libro o DVD, o dar de alta/baja simultáneamente a la estructura *articulo* y al *libro/DVD*. La estructura *articulo* resultante sería:

```
enum tipo { libro, DVD };
union descripcion {
      struct libro milibro;
      struct DVD miDVD;
};
struct articulo
{
      enum tipo mitipo;
      union descripcion midescripcion;
      float precio;
      unsigned int existencias;
};
```

Y entonces la jerarquía de las librerías de nuestra aplicación sería completamente diferente:

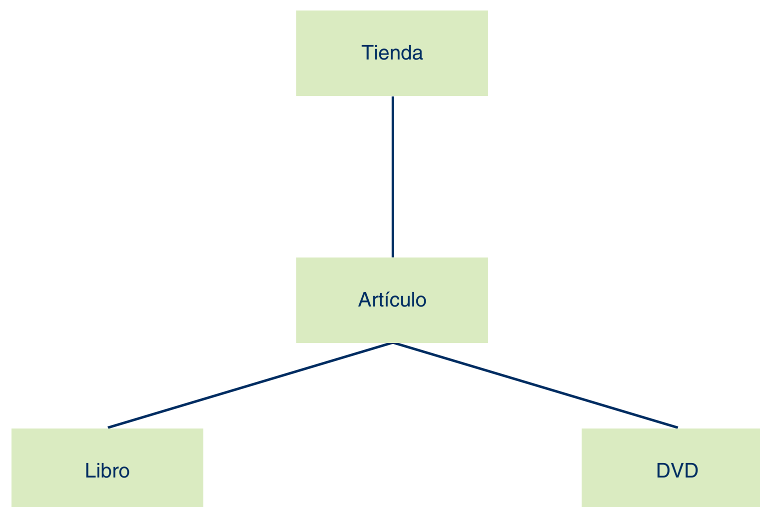


Fig. 5.4. *Jerarquía de las librerías de la aplicación.*

A partir de ahora supondremos que, en este ejemplo, optamos por la primera de las dos asociaciones posibles: artículos y libros/dvd. Empezaremos por definir el archivo de declaración articulo.h con la estructura *articulo* y la cabecera de las funciones que implementa:

```
struct articulo
{
      char cod_art[10];
      float precio;
      unsigned int existencias;
};
```

```
float Vender( struct articulo *ar, int cantidad );
struct articulo Alta(long int codigo, float euros, unsigned int cantidad );
void Visualizar(struct articulo *ar );
void Reponer(struct articulo *ar, int cantidad );
```

La correspondiente implementación de estas funciones en el fichero articulo.c quedaría:

```
#include <stdio.h>
#include <string.h>
#include "articulo.h"

struct articulo Alta(char *codigo, float euros, unsigned int cantidad )
{
      struct articulo ar;
      ar.cod_art[0] = codigo;
      ar.precio = euros;
      ar.existencias = cantidad;
      return ar;
}
void Visualizar(struct articulo *ar )
{
      printf("Código del artículo: %s\n", ar->cod_art);
      printf("Precio del artículo: %f\n", ar->precio);
      printf("Existencias del artículo: %d\n", ar->existencias);
}
float Vender( struct articulo *ar, int cantidad )
{
      ar->existencias= ar->existencias-cantidad;
      return (cantidad*ar->precio);
}
void Reponer(struct articulo *ar, int cantidad )
{
      ar->existencias= ar->existencias+cantidad;
}
```

A partir de esta librería básica de funciones podemos implementar nuevos procedimientos que den mayores funcionalidades. Por ejemplo, supongamos que una vez introducido un artículo no puede ser introducido otro que tenga el mismo código. Para realizar una función que verifique si es válido o no el nuevo artículo, podemos utilizar la función de comparación:

```
int Valida( struct articulo *ar1, struct articulo *ar2 )
{

      if(!strcmp( ar1->cod_art, ar2->cod_art ))
        return 0;
      else return 1;
}
```

```
/* Programa con función Valida */
main()
{
      int i;
struct articulo  ar[5], br[5];
      ...
      /* Carga de los arrays con datos */
      ...

      for (i=0; i<5; i++){
        if (!Valida(ar[i], br[i]))
                printf ("Los artículos de la posición %d son
                        iguales\n", i+1);
        else
                printf ("Los artículos de la posición %d son
                        distintos\n", i+1);
      }
      ...
      /* Carga de los arrays con nuevos datos */
      ...

      for (i=0; i<5; i++){
        if (!Valida(ar[i], br[i]))
                printf ("Los artículos de la posición %d son
                        iguales\n", i+1);
        else
                printf ("Los artículos de la posición %d son
                        distintos\n", i+1);
      }

}

/* Programa sin función Valida */
void main()
{
      int i;
      struct articulo  ar[5], br[5];
      ...
      /* Carga de los arrays con datos */
      ...

      for (i=0; i<5; i++){
        if (!strcmp(ar[i]->cod_art, br[i] ->cod_art))
                printf ("Los articulos de la posición %d son
                        iguales\n", i+1);
        else
        printf ("Los articulos de la posición %d son
                distintos\n", i+1);
      }
```

```
       ...
       /* Carga de los arrays con nuevos datos */
       ...

       for (i=0; i<5; i++){
         if (!strcmp(ar[i]->cod_art, br[i] ->cod_art))
         printf ("Los artículos de la posición %d son iguales\n",
                   i+1);
         else
         printf ("Los artículos de la posición %d son distintos\n", +1);
       }

}
```

Supongamos que se utiliza la función *Valida* en muchos lugares del programa y que es necesario hacer un cambio que implica la modificación de la condición de la misma. Este cambio en un único punto de la librería modifica automáticamente el funcionamiento de todo el programa. En la versión con función *Valida* sólo hay que modificar el código de la misma para que el programa quede actualizado, no siendo necesario modificar el programa principal que la invoca. Si esta función no existiera, el programador debería ir buscando en el programa cada uno de los lugares donde se ha utilizado la condición de comparación y añadirle la condición de desigualdad de existencias, lo cual lleva mucho más tiempo y hará probablemente que se produzca algún error, al pasar inadvertida la modificación en alguna sentencia. Ese tipo de errores son muy difíciles de depurar.

```
int Valida( struct articulo *ar1, struct articulo *ar2 )
{

       if(!strcmp( ar1->cod_art, ar2->cod_art ) && (ar1->existencias ==
         ar2-> existencias))
         return 0;
       else return 1;

}

/* Programa con función Valida */
main()
{
       int i;
       struct articulo  ar[5], br[5];
       ...
       /* Carga de los arrays con datos */
       ...

       for (i=0; i<5; i++){
         if (!Valida(ar[i], br[i]))
         printf ("Los artículos de la posición %d son iguales\n",
                   i+1);
         else
         printf ("Los artículos de la posición %d son distintos\n", +1);
       }
```

```
        ...
        /* Carga de los arrays con nuevos datos */
        ...

        for (i=0; i<5; i++){
          if (!Valida(ar[i], br[i]))
          printf ("Los artículos de la posición %d son iguales\n", +1);
          else
                    printf ("Los artículos de la posición %d son
                              distintos\n", i+1);
        }

}

/* Programa sin función Valida modificado*/
main()
{
        int i;
        struct articulo  ar[5], br[5];
        ...
        /* Carga de los arrays con datos */
        ...

        for (i=0; i<5; i++){
          if (!strcmp(ar[i]->cod_art, br[i] ->cod_art) && (ar1->
                    existencias == ar2-> existencias))
                    printf ("Los artículos de la posición %d son
                    iguales\n", i+1);
          else
                    printf ("Los artículos de la posición %d son
                    distintos\n", i+1);
        }
        ...
        /* Carga de los arrays con nuevos datos */
        ...

        for (i=0; i<5; i++){
          if (!strcmp(ar[i]->cod_art, br[i] ->cod_art) && (ar1->
                    existencias == ar2-> existencias))
          printf ("Los artículos de la posición %d son
                    iguales\n", i+1);
          else
                    printf ("Los artículos de la posición %d son
                    distintos\n", i+1);
        }
}
```

La ocultación de la información no sólo es útil cuando se modifica alguna función, sino que también permite la modificación de la estructura completa, sin necesidad de modificar el programa principal.

Imagina que queremos cambiar la estructura porque los artículos no son unitarios (un libro por entregas o una serie en DVD con varios capítulos). Entonces, las ventas no serán unitarias por lo que el campo existencias y las ventas no serán números enteros, sino valores reales. Con la librería de funciones básicas que hemos realizado no es necesario cambiar ni una línea de código del programa principal. Únicamente debemos redefinir parte de la librería para que la estructura defina dicho campo de forma apropiada y las funciones *alta* y *vender* reciban un *float* como argumento cantidad, y la función *Visualizar* utilice *%f* en lugar de *%d* en la instrucción *printf.*

El ejemplo que vamos a desarrollar en este apartado consiste en crear una biblioteca que permita representar en un plano de dos dimensiones el seguimiento de las trayectorias de los aviones en un determinado espacio aéreo. Para ello, debemos representar fundamentalmente tres estructuras:

a) Las zonas del espacio aéreo que están restringidas (por saturación, razones de seguridad o uso militar).

b) Las trayectorias que siguen los aviones.

c) El área concreta del mapamundi sobre la que nuestra aplicación muestra el seguimiento.

Ejemplo práctico

4 Aunque la aplicación pretende simular la trayectoria de unos aviones en un espacio del mundo, esta simulación es una simplificación, ya que el mundo real tiene tres dimensiones (incluyendo los aviones y las zonas restringidas a la navegación aérea comercial). Para nuestro ejemplo sólo nos interesa una visión en dos dimensiones, en la que la altitud de los aviones y de las zonas restringidas no queda reflejada y la ignoramos a pesar de su importancia real.

Asimismo, el planeta tiene unas dimensiones suficientemente grandes como para que no sea útil mostrar las trayectorias de los aviones a una escala global, por lo que debemos limitar nuestro seguimiento a una parte de todo el planeta. Para saber qué parte del mundo queremos mostrar, utilizaremos una estructura dedicada a este fin.

Hemos comentado con anterioridad que otra de las estructuras que nos interesa representar es la zona restringida, de forma que refleje su extensión (en forma de un polígono en 2D) y las razones de dicha restricción (seguridad, militar, saturación).

Adicionalmente, hemos de representar a las aeronaves y su movimiento. Con este objeto, la estructura que las representa almacena la velocidad a la que se desplazan. La trayectoria que siguen se supone prefijada en un fichero antes de comenzar la ejecución de la simulación (pero con trayectoria independiente para cada uno de los aviones).

En resumen, la aplicación que queremos construir deberá permitir representar una zona del planeta con dos tipos de elementos: las zonas restringidas y los aviones en movimiento. Los aviones no podrán moverse por las zonas restringidas. Para establecer la configuración inicial de este mundo, utilizaremos lectura de ficheros escritos en modo texto (para una mayor comprensión y sencillez). Así, el programa principal de nuestra aplicación utilizará las funciones de las librerías que definamos con objeto de proporcionar al usuario las siguientes operaciones:

a) Modificar las condiciones de ejecución: el fichero que define el mundo, la velocidad a la que transcurre el tiempo para los movimientos y el tiempo total que durará la simulación.

b) Ejecutar la simulación del seguimiento.

c) Reiniciar la simulación del seguimiento.

Adicionalmente también se podría enriquecer nuestra aplicación con un editor mediante el cual se pudiera modificar el fichero que almacena la definición del mundo.

Para conseguirlo agruparemos las distintas funciones en tres módulos o librerías: una dedicada a la lectura de ficheros, otra a mostrar en pantalla el estado de la zona que estamos siguiendo y una tercera para realizar los movimientos de los aviones.

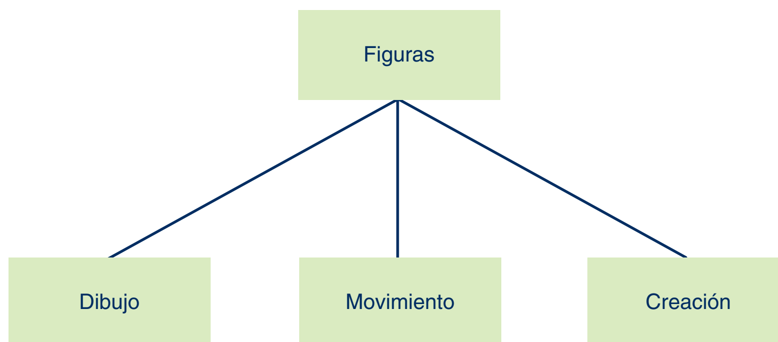


Fig. 5.5. *Agrupación de los tres módulos.*

Uno de los tipos de datos en los que se fundamenta nuestra aplicación es el que representa un punto en coordenadas polares. La posición de un avión se representa mediante un punto de este tipo y la de una zona restringida con una secuencia de puntos dinámicamente enlazados que define un polígono.

```
struct punto
{
      double r, angulo;
      struct punto *siguiente;
};
```

Otra de las estructuras básicas de la aplicación es la que permite definir la trayectoria de un avión. Para ello, suponemos que un avión tiene dos motores (uno por cada ala) y que su dinámica se define por las velocidades que imprimen.

Como se explicó anteriormente, los aviones seguirán trayectorias definidas *a priori* en un fichero. Estas trayectorias vienen definidas por tramos en los que se mantiene una determinada aceleración producida por el motor de cada ala del avión. Así, utilizaremos una estructura enlazada para encadenar una trayectoria con la siguiente, de la forma indicada a continuación:

```
struct trayectoria
{
      double tramo, adcha, aizda;
      struct trayectoria *siguiente;
}
```

Con estas dos estructuras ya es posible definir las partes en común de todas las figuras: los aviones, las zonas restringidas y el campo de visión de la pantalla con la que mostramos el seguimiento.

Para ello, supondremos que los puntos que delimitan el polígono correspondiente a una zona restringida o al campo de visión se definen en coordenadas polares con respecto al centro de la zona restringida o del campo de visión. En cambio, a los aviones se les supone sin grosor y, por tanto, estarán representados únicamente mediante su punto central.

Por último, mencionar que utilizaremos una estructura doblemente enlazada (con la figura siguiente y la anterior) para ir recorriendo todas las figuras que se presenten en una ejecución de nuestro programa. Así, la parte común a los tres tipos de figuras es la siguiente:

```
enum tipo { campo_vision, zona, avion };
struct figura
{
      enum tipo mitipo;
      double x,y;
      ...
      struct figura *anterior;
      struct figura *siguiente;
}
```

La parte ausente en el código anterior es la correspondiente al movimiento del avión, que vendrá definido por su velocidad *(v)* y su velocidad angular *(w)*, su distancia entre las ruedas *(r)* y la dirección que lleva el avión *(tetha)*. Debido a que estos campos sólo son aplicables a los aviones y, en cambio, la secuencia de puntos definida por el atributo *mispuntos* es aplicable a los otros dos tipos de figuras (zonas restringidas y campo de visión), por tanto se pueden representar como una unión semejante a la siguiente:

```
struct velocidad
{
      double v,w,r,tetha;
      struct trayectoria *mistrayecs;
}
union grosor_o_velocidad
{
      struct velocidad mivelocidad;
      struct punto *mispuntos;
}
```

Y así la estructura completa de figura quedaría:

```
enum tipo { campo_vision, zona, avion };
struct figura
{
      enum tipo mitipo;
      double x,y;
      union grosor_o_velocidad no_comun;
      struct figura *anterior;
      struct figura *siguiente;
}
```

Gráficamente vamos a ver un ejemplo de zona restringida y de avión en movimiento.

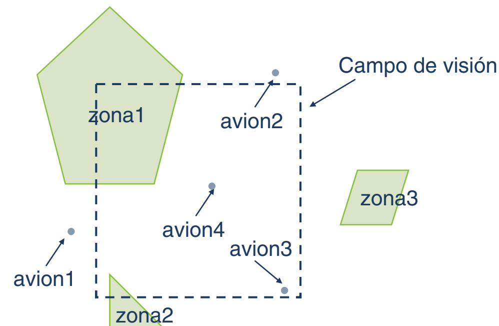


Fig. 5.6. *Zonas restringida y de avión en movimiento.*

Antes de pasar a detallar el funcionamiento interno de las funciones, haremos una breve síntesis de las funciones que vamos a agrupar en cada uno de los tres módulos:

a) Creación y destrucción de figuras:

En esta librería se especifican todas las funciones que crean las estructuras para los datos, leen dichos datos de un fichero y vacían y destruyen las estructuras creadas.

- *LeerFiguras:* esta función crea la lista de figuras y llama a la función *LeerFigura*, que lee una figura tantas veces como sea necesario para definir todo el entorno.
- *LeerFigura:* esta función simplemente va llamando a todas las funciones que leen los campos que definen una Figura.
- *LeerTipo:* lee el tipo de la figura; los validos son: campo de visión, avión y zona restringida.
- *LeerNodos:* crea la estructura de los puntos de una figura. Al menos debe existir un punto por figura (es el caso de los aviones).
- *LeerInicial:* lee los valores de la posición inicial de la figura. Éstos son las coordenadas *x*, *y*. En el caso de los aviones también se leerán los valores de *v*, *w*, *r* y *tetha*.
- *LeerTrayecs:* crea la lista de trayectorias y las lee desde el fichero.
- *LeerControl:* lee un carácter del fichero y comprueba si es el carácter de control esperado; si lo es, se salta el resto de la línea llamando a la función *LineaSiguiente;* en otro caso devuelve error.
- *LineaSiguiente:* lee caracteres del fichero hasta encontrar un fin de línea (respetando el fin de fichero), es decir, ignora el resto de la línea.
- *LeeNumero:* lee caracteres del fichero hasta encontrar uno que sea distinto del espacio en blanco y devuelve el numero correspondiente (*float* o *int*).
- *BorrarFiguras:* libera toda la memoria ocupada por la lista de figuras que compone el mundo en 2D.

- *BorrarNodos:* libera toda la memoria ocupada por la lista de puntos que delimita una región prohibida o un campo de visión.

- *BorrarTrayecs:* libera toda la memoria ocupada por la lista de tramos que define la trayectoria de un avión.

b) Movimiento de figuras (sólo aviones)
En este módulo se definen las funciones que nos permiten mover los aviones (son las únicas figuras que tienen dicha capacidad).

- *MueveFigura:* los aviones describen una serie dada de trayectorias en función del instante en que nos encontremos (esto viene dado en el fichero de entrada que describe el mundo). Además, cada una de estas trayectorias tiene una aceleración especifica para cada rueda. Con estas aceleraciones correspondientes al tramo del instante actual se calcula la nueva velocidad.

- *AplicaMov:* con la nueva velocidad ya calculada se aplica sobre la posición y dirección del avión para determinar su nueva posición y velocidad.

c) Dibujo de figuras

Esta librería incluye la definición de las funciones que manejan la presentación en pantalla del seguimiento de los aviones. A continuación, se indican todas las funciones que deberían componer esta librería, aunque por razones de espacio en este capítulo sólo se han desarrollado *IniciaG, CierraG, DibujaFigura* y *EntraFigura*.

- *IniciaG:* función que inicializa las funciones gráficas.

- *CierraG:* función que cierra el modo gráfico.

- *DibujaMundo:* presenta en la pantalla las zonas restringidas y los aviones que se han leído del fichero y cuyas coordenadas se encuentran dentro de los límites del campo de visión. Dibuja también el borde del campo de visión.

- *Corte:* calcula matemáticamente los puntos de corte de una recta con los lados del campo de visión.

- *DibujaFigura:* va dibujando un avión o una zona uniendo sus vértices y comprobando si están dentro de la campo de visión o no.

- *EntraFigura:* comprueba si la figura está dentro de los limites de la pantalla, fuera o bien parte dentro y parte fuera.

- *Scroll:* función que permite modificar los límites del campo de visión que se muestra en pantalla.

- *DibujaCampo:* dibuja las cuatro líneas que limitan el campo de visión.

- *BorraCampo:* oculta las figuras del campo de visión dibujando un rectángulo negro con las dimensiones del mismo.

- *BorraFigura:* borra una figura del campo de visión.

- *CorteLinea:* comprueba si algún punto de la línea se ha cruzado con alguna zona restringida o avión. Esto lo hace viendo si en los puntos en los que debe dibujar la línea ya había algo pintado.

De estas tres librerías, únicamente vamos a ver cómo se realizarán los cálculos necesarios para implementar el movimiento de los aviones. Para llevar a cabo estos cálculos tenemos en cuenta las siguientes consideraciones del problema:

- El avión esta dotado de dos motores, uno en cada ala sobre las cuales se puede aplicar una determinada velocidad lineal (v1 y v2).
- Dichas alas no pueden girar, con lo que para que el avión pueda girar lo que hacemos es aplicar una velocidad mayor o menor al motor de un ala que al de la otra.
- Las alas están separadas por una distancia r.
- El avión en un determinado instante se encuentra en una posición (X,Y) y con una dirección θ con respecto del eje vertical.

Existen dos posibles movimientos diferentes que puede describir un móvil:

- **Traslación.** Desplazamiento definido por la velocidad lineal *v:*

```
Posición = Velocidad * Tiempo
```

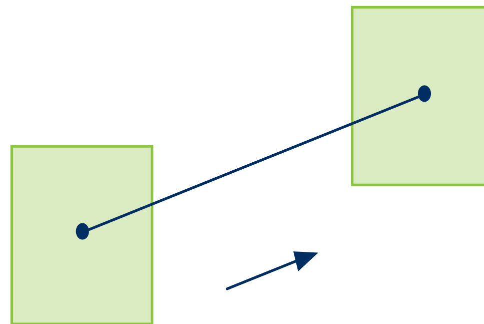


Fig. 5.7. *Desplazamiento definido por la velocidad lineal.*

- **Rotación.** Movimiento sobre sí mismo definido por la velocidad angular *w:*

```
Ángulo = Velocidad Angular * Tiempo
```

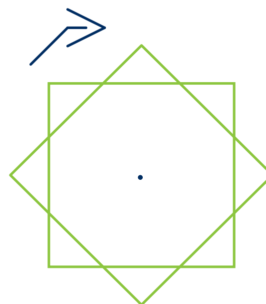
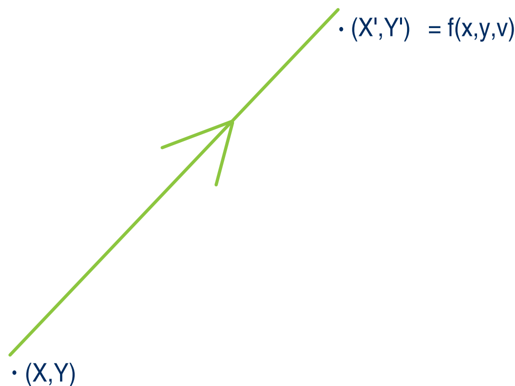


Fig. 5.8. *Movimiento sobre sí mismo definido por la velocidad angular w.*

Así pues lo que varía en cada tipo de movimiento es:

1. Su nueva posición *(X', Y')* es función de la posición en la que se encontraba antes y de la *v:*



2. Su dirección θ que es función de la antigua orientación θ y de la velocidad angular aplicada *w:*

Así, las fórmulas a aplicar son las siguientes:

- La velocidad del avión en un determinado instante es la media de las velocidades aplicadas a sus ruedas:

```
v(n) = (vdcha(n)+ vizda(n)) /2
```

- La posición de un avión en un determinado instante se calcula sumando la posición que tenía en el instante anterior con la velocidad que lleva por el tiempo.

  Tenemos que tener en cuenta también que el avión se mueve en cualquier dirección, con lo que en unos casos la nueva coordenada *X* o *Y* aumenta o disminuye en función de la dirección θ en la que se mueva, con lo que tendremos que considerar el seno en el caso de la *X* y el coseno en el caso de la *Y:*

$X(n+1)=X(n)+v(n)\cdot sen\theta\cdot t$

$Y(n+1)=Y(n)+v(n)\cdot cos\theta\cdot t$

- La velocidad angular en un determinado instante se calcula, de la misma forma que en el caso lineal, a partir de las velocidades lineales de las alas pero considerando la diferencia y el radio entre ellas:

  $\omega(n) = (vdcha(n) + vizda(n)) / r$

- La dirección en la que se mueve el avión en un determinado instante se calcula a partir de la dirección que tenía en el instante anterior y de la velocidad angular del instante anterior:

  $\theta(n+1)=\theta(n) + \omega(n)\cdot t$

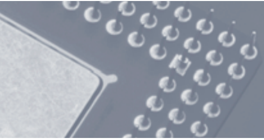
La librería que modeliza este movimiento se compone de dos funciones: en una se calcula el tramo en el que nos encontramos y obtiene las velocidades que deseamos aplicar, y en otra se calculan los nuevos valores de la situación actual de la figura definida por las coordenadas *x* e *y*, y por la orientación del avión determinada por el ángulo *tetha*.

A continuación, se muestra el código de la primera de ellas, que recibe como argumento el tiempo actual (con el que seleccionaremos el tramo en vigor) y el tiempo transcurrido desde la última actualización de la posición:

```
void MueveFigura( struct figura *f, float Ttotal, float Tparcial )
{
       float nueva_v, nueva_w;
       struct trayectoria *tramo;
       tramo = f-> nocomun.mivelocidad.mistrayecs;
       while(tramo->siguiente != 0 && tramo->tramo < Ttotal)
         tramo = tramo->siguiente;
       nueva_v = ( tramo->adcha + tramo->aizda ) / 2.0;
       nueva_w = ( tramo->aizda - tramo->adcha ) /
                   f->nocomun.mivelocidad.r;
       AplicaMov( f, na, nalfa, Tparcial );
}
```

Esta función primero calcula la nueva *v* y *w* del tramo actual, y una vez obtenidas estas velocidades se llama a la función *AplicaMov* que modifica los valores de posicionamiento de la figura. La función que mueve la figura es entonces:

```
void AplicaMov( struct figura *f, float nv, float nw, float Tparcial )
{
       f->nocomun.mivelocidad.v = nv;
       f->x =  f->x + ( f->v * sin(
         f->nocomun.mivelocidad.tetha)* tf );
       f->y =  f->y + ( f->v * cos(
         f->nocomun.mivelocidad.tetha) * tf );
         f->nocomun.mivelocidad.w =  nw;
         f->nocomun.mivelocidad.tetha =
         f->nocomun.mivelocidad.tetha +
```




```
        f->nocomun.mivelocidad.w * tf;
      if( f->nocomun.mivelocidad.tetha > 2 * 3.1416 )
        for(;(f->nocomun.mivelocidad.tetha =
                f->nocomun.mivelocidad.tetha - 2 * 3.1416)
                > 2 * 3.1416;);
      else if( f->nocomun.mivelocidad.tetha < 0 )
        for(;(f->nocomun.mivelocidad.tetha =
                f->nocomun.mivelocidad.tetha + 2 * 3.1416)
                < 0;);
}
```

Las líneas finales de esta función tienen por objeto impedir que el ángulo *tetha* sea mayor que *2 * pi*. En dicho caso, habrá dado una vuelta completa y debe tomar los valores próximos a cero que correspondan al giro.

Ejercicios

1. Crea una librería de funciones para operar con números complejos.

2. Desarrolla las funciones nombradas en este capítulo para jugar al ajedrez.

3. Genera un programa que haciendo uso de la librería artículo explicada en este capítulo permita realizar una búsqueda de un producto y su posterior compra, completando la implementación de las funciones de las librerías de libros y DVD cuya declaración ha sido indicada en el texto.

4. Implementa las funciones, la librería de creación y destrucción de figuras correspondiente al ejemplo de seguimiento de aviones en el espacio aéreo.

Evaluación

1. Explica por qué razón las librerías se definen mediante dos ficheros, uno de extensión .h y otro .c.
2. Describe qué tienen en común la visión clásica y la moderna de la programación estructurada.
3. Enumera las ventajas de la programación estructurada sobre la programación tradicional.

Actividades prácticas

1. **Desarrolla una librería de funciones más que permita aviones inteligentes, esto es, que su movimiento no esté prefijado mediante tramos, sino que sea capaz de moverse por el mundo que dibujemos según una lógica de control (por ejemplo, evitando las zonas restringidas y otros aviones). Para ello, se deberá dotar a dicha figura de un destino que pretende alcanzar y de sensores que le permitan percibir el mundo.**

2. **Crea otra librería de funciones que permita que existan figuras móviles controladas por el usuario. Esto es, que mediante cuatro teclas (arriba, abajo, izquierda y derecha) permita a un usuario guiar una figura móvil por el mundo.**

3. **Propón una ampliación al ejemplo de gestión de productos en una tienda desarrollando las librerías asociadas a las otras estructuras necesarias para la aplicación: Proveedores, Clientes, Empleados y Pedidos.**