Turbo Vision

con

C++

Francisco Charte Ojeda

INTRODUCCION

Antes de comenzar a estudiar el uso de Turbo Vision es necesario conocer qué estamos utilizando y cual es su finalidad, para a continuación ir entrando en detalles.

¿Qué es Turbo Vision?

Turbo Vision es una jerarquía de clases C++ especializada en la creación de interfaces de usuario. A partir de esta jerarquía de clases nosotros podemos crear objetos o derivar nuevas clases, lógicamente ajustándolas más a nuestras necesidades.

Como sabemos, antes de poder crear un objeto de una clase es necesario haberla definido, aunque la implementación de métodos puede encontrarse en otro módulo o librería. Previo al uso de cualquier clase Turbo Vision, es necesario incluir en nuestro código el archivo de cabecera **TV.H**, que contiene las directivas necesarias para incluir la definición de las clases que vayamos a utilizar. Lógicamente antes tendremos que comunicar de algún modo qué clases son esas. Para ello definiremos, por cada clase que utilicemos, el símbolo **Uses_xxx**, donde **xxx** representa el nombre de la clase.

Elementos de una aplicación Turbo Vision

En Turbo Vision una aplicación se compone básicamente de **vistas** y **motor**, que responden a **eventos** externos o internos.

Las vistas de una aplicación son elementos tales como la barra de menú, las cajas de diálogo donde se da entrada a los datos, la línea de estado o la ventana de ayuda. En definitiva las vistas es aquello con lo que interactúa el usuario, ellas forman el interfaz. Podríamos decir que las vistas son el cristal delantero de un coche, sus espejos retrovisores y todo el panel que nos informa sobre el estado actual. Aunque cambiemos de coche estos elementos siempre son los mismos, y tienen habitualmente los mismos diseños. De igual forma, el interfaz de Turbo Vision hará que nuestras aplicaciones tengan todas una misma apariencia y funcionamiento, que es lo más lógico.

El motor de una aplicación esta compuesto por todos aquellos procesos que tratan la información que entra el usuario para obtener unos resultados. Básicamente el motor es lo que diferencia a una aplicación de otra, de igual forma que a un coche. Aunque todos los coches se controlen igual y tengan las mismas vistas, el motor hace que cada uno esté enfocado a un fin distinto, deportivo, familiar, etc. En el caso de una aplicación, el motor decidirá su finalidad.

Los eventos son acciones externas o internas que pueden introducir información o modificar el comportamiento de la aplicación. En el caso de los coches existen básicamente dos tipos de eventos: los de los pedales y el volante. En nuestro programa los eventos serán enviados por el teclado y por el ratón principalmente.

El esqueleto de aplicación TApplication

La clase **TApplication**, o una derivada suya, será siempre el comienzo de cualquier aplicación Turbo Vision. Un objeto de esta clase es es realidad una vista más, pero que contiene a su vez otras vistas. Podríamos decir que un objeto **TApplication** es un grupo que contiene objetos tales como la barra de menú, la línea de estado y el despacho, que son en sí vistas u otros grupos.

Todas las vistas Turbo Vision están derivadas de una u otra forma de la clase **TView**. Cualquier objeto de esta clase sabe como representarse a sí mismo en pantalla, y como responder a los eventos que se produzcan. Algunas clases, como es el caso de **TApplication**, derivan de **TGroup**, que a su vez deriva de **TView**, teniendo así la posibilidad de contener otras vistas. Un objeto derivado de **TGroup** se representa a sí mismo en pantalla en base a los objetos que contiene, y gestiona los eventos recibidos pasándolos a estos objetos.

Una aplicación Turbo Vision es así de simple, o de compleja, según se mire. Un objeto se representa en pantalla haciendo que los objetos que contiene se representen en pantalla, objetos que a su vez pueden ser grupos y contener otros. Este mismo objeto responde a los eventos pasándolos a los objetos que contiene, hasta que alguno de ellos lo gestione. Sin embargo, este es el esquema de una aplicación vacía, ya que aunque recoge información y hace funcionar el interfaz de usuario no produce ningún resultado. A esto es a lo que se llama un marco o esqueleto de aplicación, y es lo que nos ofrece Turbo Vision, una plantilla genérica donde incluyendo los procesos que forman el motor obtengamos una aplicación final.

LA APLICACION MAS SIMPLE

Sin duda el ir viendo resultados a medida que se estudia es algo que estimula y anima a seguir. Por ello, y aunque no obtengamos nada útil, vamos a ver una primera aplicación en funcionamiento, la aplicación más simple posible.

Creación de un objeto TApplication

La clase **TApplication** tiene, entre otros, un miembro llamado **run**(), que tiene como finalidad poner en funcionamiento la aplicación creada. Aunque en un principio podría parecer tan simple como crear un objeto **TApplication** y enviar un mensaje al método **run**(), esto no funciona, puede comprobarlo usted mismo.

El constructor y destructor definidos para la clase **TApplication** son protegidos, lo que quiere decir que sólo son accesibles desde la misma clase y sus derivadas, pero no desde un objeto externo. O sea, que si intentamos crear un objeto de esta clase obtendremos un error indicándonos que resulta imposible acceder a su constructor y destructor, pasos sin los cuales no se puede proceder a la creación del objeto.

Derivación de una clase propia

Por lo tanto tendremos que crear una clase derivada de **TApplication**, en la que el constructor esté en la parte pública, permitiendo así crear un objeto sin problemas.

Nuestra clase, a la que podríamos llamar **AplicacionEstandar**, heredará, además de los miembros de **TApplication**, todos los de las clases base que en la jerarquía están como ancestros de **TApplication**. Es decir, heredará los miembros de **TProgram**, **TProgInit**, **TGroup**, **TView**, etc. Hay que prestar especial atención a los constructores, ya que para llegar a crear un objeto de la clase **AplicacionEstandar** tendrán que ejecutarse todos los constructores de las clases base, para que cada una construya su porción del objeto. La mayoría de estas clases base tienen un constructor por defecto, lo que significa que no es necesario hacer nada especial para que al crear un objeto sean ejecutados. Sin embargo, el constructor de la clase **TProgInit** necesita tres parámetros:

1 - La dirección de un método miembro que se encargue de crear la línea de estado. Nuestra clase hereda de **TProgram** el método **initStatusLine**, que es el método normalmente usado para este fin.

2 - La dirección de un método miembro que procese la creación de una barra de menús. También a través de **TProgram** nuestra clase hereda un método con esta finalidad, denominado **initMenuBar**.

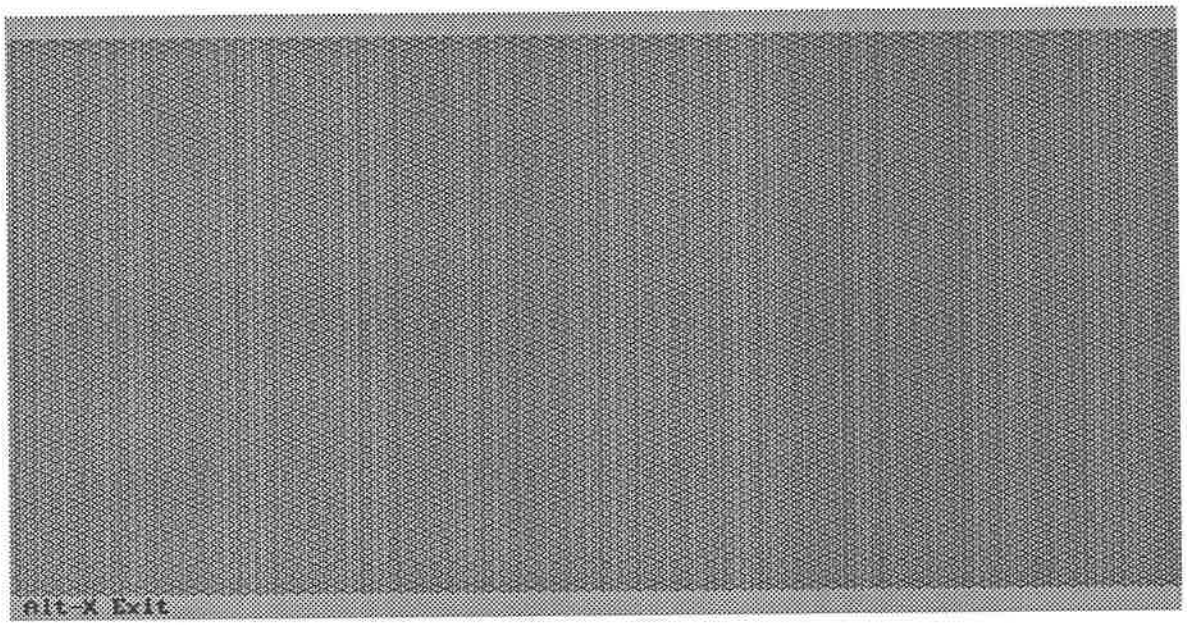
3 - La dirección de un método miembro que se encargue de crear el despacho o fondo de pantalla sobre el que irán apareciendo el resto de las vistas. El método heredado de **Tprogram** con este fin se llama **initDeskTop**.

En principio, puesto que hemos heredado estos tres miembros, no es necesario crear métodos específicos para estas funciones, pero como veremos más adelante es lo más lógico, porque distintas aplicaciones tendrán distintas opciones de menú y distintas líneas de estado.

Conociendo esto no necesitamos nada más para crear nuestra primera aplicación Turbo Vision. En el listado **SEMTV01.CPP** puede ver como se crea la clase **AplicacionEstandar** derivando de **TApplication**. El constructor de esta nueva clase pasa al de **TprogInit** las direcciones de los métodos encargados de inicializar la línea de estado, la barra de menú y el despacho. Por último, ya en **main**(), se crea una instancia de nuestra clase y se ejecuta por medio del método **run**().

```
//
// SEMTV01.CPP                Un primer ejemplo con Turbo Vision
//

#define Uses_TApplication      // Se definen las clases que se van a usar

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

//
// Se deriva una clase a partir de TApplication,
// a la que se llama AplicacionEstandar
//

class AplicacionEstandar : public TApplication
{
        public :
                // Se define el constructor de la clase

                AplicacionEstandar() :
                        TProgInit( & initStatusLine,
                                & initMenuBar,
                                & initDeskTop )
                        {}
};

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```

Una vez compilado (no olvide activar en la opción **Linker|Libraries** la inclusión de la librería Turbo Vision) podrá ejecutarlo. En pantalla aparecerá el despacho, un fondo uniforme de color gris, con una barra de menú sin opciones y una línea de estado con un sólo mensaje, la combinación Alt-X abandona la aplicación.



SEMTV01

PERSONALIZACION DE LA APLICACION

Está claro que una aplicación con una barra de menú vacía y una línea de estado indicando que la pulsación Alt-X abandona no es muy útil. Necesitamos que en la barra de menú aparezcan las opciones de nuestro programa, y en la línea de estado la ayuda o indicaciones necesarias. Vamos a dar estos primeros pasos en la construcción de nuestra aplicación.

Diseño de la línea de estado

El método encargado de inicializar la línea de estado es **initStatusLine**, que como hemos visto antes nuestra clase hereda desde **TProgram**. Este método está pensado para ser redefinido en cada nueva clase de aplicación, estableciendo el tamaño y posición de la línea de estado, los comandos activos y el texto que ha de aparecer en ella. Este método, al igual que **initMenuBar** e **initDeskTop**, es estático, existiendo una sóla instancia independiente del número de objetos que creemos de nuestra clase.

initStatusLine recibe como parámetro un objeto de la clase **TRect**, especificando el área total de trabajo disponible, a partir de la cual habrá que decidir el área dedicada a la línea de estado. Un objeto **TRect** está compuesto de dos objetos **TPoint**, uno llamado **a** y otro llamado **b**, definiendo el primero el punto correspondiente a la esquina superior izquierda del área y el segundo el de la esquina inferior derecha. Cada uno de estos objetos **TPoint** está formado por dos enteros, llamados **x** e **y**. Con esta información tendremos que crear un objeto de la clase **TStatusLine**, una línea de estado, y devolver un puntero a él. En el listado SEMTV02.CPP puede ver el proceso de creación de este objeto.

La construcción del objeto **TStatusLine**, por medio del operador estándar *new*, puede parecer en principio algo compleja, ya que se utilizan otras dos nuevas clases: **TStatusDef** y **TStatusItem**. El constructor de la clase **TStatusLine** recoge dos parámetros, un objeto **TRect** definiendo el área donde se mostrará la línea de estado, habitualmente la última línea de pantalla, y un objeto **TStatusDef**, especificando los códigos de ayuda disponibles en cada momento y las campos a mostrar en la línea de estado. En realidad este segundo parámetro es una lista enlazada, construida por el operador sobrecargado +, de objetos **TStatusDef**, cada uno de los cuales contiene a su vez una lista enlazada de objetos **TStatusItem**. Pero vayamos paso a paso.

```
//
// SEMTV02.CPP                  Creación de una línea de estado a medida
//

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

//
// Se deriva una clase a partir de TApplication,
// a la que se llama AplicacionEstandar
//

class AplicacionEstandar : public TApplication
{
    public :

                // Se define el constructor de la clase

        AplicacionEstandar() :

            TProgInit( & initStatusLine,
                       & initMenuBar,
                       & initDeskTop )

            {}

                // Método para inicializar la línea de estado

        static TStatusLine * initStatusLine( TRect ) ;
} ;

    // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
            // Se define como área de línea de estado la última línea
            // del área total

    Area . a . y = Area . b . y - 1 ;

            // Se crea una línea de estado y se devuelve

    return new TStatusLine( Area,
                            * new TStatusDef( 0, 65535 ) +
                            * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit )
                          ) ;
}

int main()
{
    AplicacionEstandar      MiAplicacion ; // Se crea un objeto

    MiAplicacion . run() ; // Y se ejecuta

    return 0 ; // Fin del programa
}
```

Normalmente la línea de estado presenta una u otra información, y cambia los comandos activos en cada momento, dependiendo del estado de la aplicación. Esto es lo que se llama ser sensible al contexto. Al crear la línea de estado definiremos, con cada objeto **TStatusDef**, los códigos de ayuda disponibles y la línea de estado a mostrar, definida por N objetos **TStatusItem**, en cada momento dependiendo del contexto. En el programa ejemplo propuesto anteriormente hay una sola línea de estado para toda la aplicación, que utiliza todos los códigos de ayuda disponibles, desde el 0 al 65535, y sólo contiene un campo, un objeto **TStatusItem**.

Cada objeto **TStatusItem** se crea a partir de un texto a visualizar en la línea de estado, el texto comprendido entre las dos tildes aparece resaltado, una combinación de teclado y un comando asociado. El texto puede ser nulo, de tal forma que simplemente se asocie

una pulsación de tecla con un comando. Tanto las combinaciones de tecla como los comandos están representados por constantes. Es posible encontrar una referencia de constantes de teclado y comandos en las páginas 213 y 466, respectivamente, del manual de Turbo Vision.

Creación de una barra de menús

El funcionamiento del método **initMenuBar**, que tendremos que redefinir para crear nuestra propia barra de menús, es similar al de **initStatusLine**. Se recibe un objeto **TRect** conteniendo el área disponible, a partir de la cual habrá que definir la posición de la barra de menú, habitualmente la línea superior de la pantalla, y crear y devolver un objeto **TMenuBar**.

El constructor de **TMenuBar** necesita dos parámetros: un objeto **TRect** definiendo el área en la que se visualizará el menú, que como he comentado anteriormente suele ser la línea superior; y un puntero a un objeto **TMenu**, que es en realidad una lista de objetos conteniendo submenús, **TSubMenu**, y opciones, **TMenuItem**. En el listado SEMTV03.CPP puede ver un ejemplo de creación de una barra de menú con dos opciones, que dan paso a dos submenús.

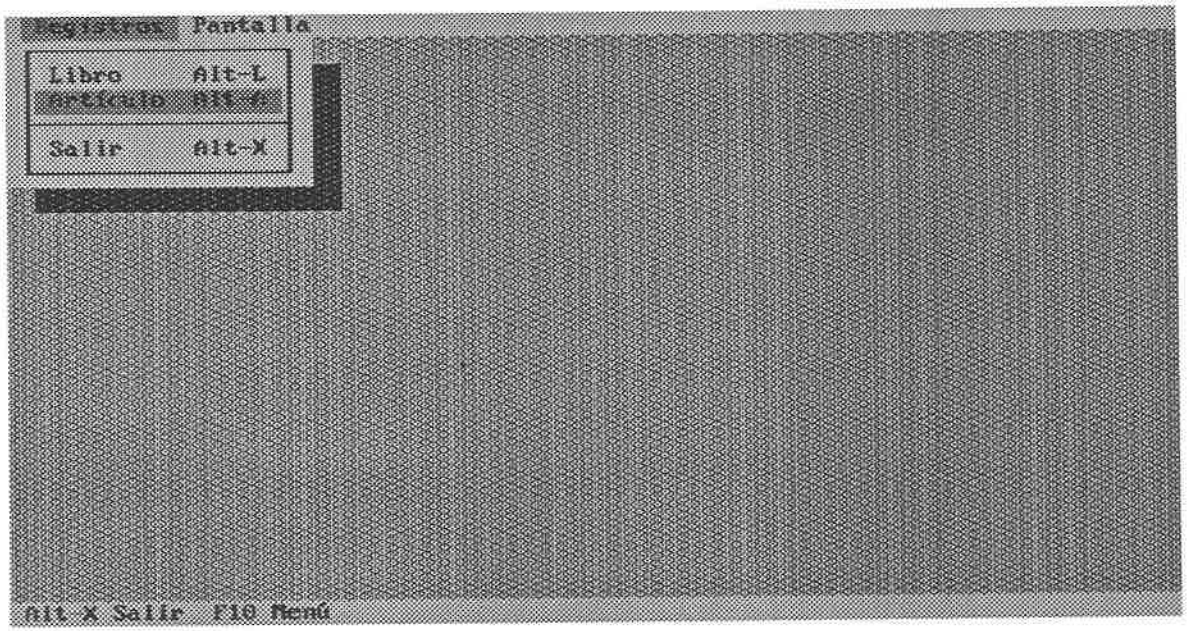
Para crear una opción en la barra creamos un objeto **TSubMenu**, especificando el título, delimitando entre tildes la tecla de selección, y la tecla de acceso rápido. Para crear opciones dentro de estos submenús creamos objetos **TMenuItem**, especificando el título, el comando que deben generar, la tecla de selección rápida, el contexto de ayuda y una descripción de la tecla de selección rápida para que aparezca junto a la opción. En principio todas las opciones no tienen porqué estar accesibles desde una tecla rápida, ni tener un contexto de ayuda.

Además de crear objetos con los dos tipos anteriores, también hemos usado el método **newLine**(), cuya finalidad es insertar una línea de separación entre opciones.

Existen varias formas de acceder a una opción de la barra de menú y a sus opciones. La forma más habitual y cómoda es utilizando el ratón. También podemos utilizar las teclas de selección rápida, que aparecen destacadas en los títulos. Si te fijas en la definición de la línea de estado verá que he añadido un nuevo comando, la pulsación de la tecla **F10** genera el comando **cmMenu**, que activa la barra de menús y nos permite utilizar los cursores para seleccionar.

```
//
// SEMTV03.CPP                  Incorporación de una barra de menús
//

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem

#include <Tv.H>         // Se incluye el archivo de cabecera Turbo Vision

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )
                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;
};
```

```
        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu )
                              ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla
        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                        * new TSubMenu( "~R~egistros", kbAltR ) +

                            * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                            * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                            newLine() +
                            * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +


                        * new TSubMenu( "~P~antalla", kbAltP ) +

                            * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                            * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                      ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```



SEMTV03

Implementación de los métodos de respuesta a eventos

En el ejemplo anterior al seleccionar cualquiera de las opciones del menú no ocurre nada, porque esta selección lo que hace es generar un comando que no está definido en Turbo Vision. Los comandos correspondientes a las opciones están definidos como constantes al principio del listado, y siguen las reglas de nomenclatura propias de Turbo Vision.

Cada vista de nuestra aplicación tiene un método, llamado **handleEvent**(), que se encarga de gestionar los eventos que se produzcan. Ya hemos visto eventos generados por el teclado, pulsaciones de tecla, por el ratón, la pulsación de un botón, y ahora vemos el tercer tipo, eventos generados por comandos. Lo que ocurre es que el gestor de eventos de nuestras vistas conoce ciertos comandos, pero no los nuevos que nosotros hemos definido. Por ello tendremos que redefinir este método, gestionando nosotros mismos los nuevos comandos.

handleEvent() recibe como parámetro un objeto **TEvent** con el evento. Este objeto contiene un campo, llamado **what**, que nos indica el tipo del evento a procesar. En la página 471 del manual es posible encontrar una tabla con los distintos tipos, designados por constantes. Según el tipo de evento utilizaremos uno de los tres objetos que forman la unión. Habitualmente nosotros siempre utilizaremos **message**, que almacena el comando generado, ya que los otros dos contienen eventos de teclado y ratón, que son procesados generalmente por las vistas sin nuestra intervención.

Al redefinir este método hay que tener en cuenta que todos los eventos que se produzcan en la aplicación pasarán por él, a pesar de que la mayoría de los eventos, producidos por teclado y ratón, deben ser procesados por el método del mismo nombre de las vistas. Por ello, el primer paso al procesar un evento será pasar éste al método **handleEvent** de nuestra clase base, **TApplication**, que se encargará de pasarlo a las demás vistas. Hecho esto tendremos que comprobar si el evento es un mensaje, y si este mensaje es uno de los comandos que hemos definido, caso en el cual tendremos que realizar la acción correspondiente a él, y que normalmente es enviar un mensaje a un método miembro. En el listado SEMTV04.CPP puede ver el método **handleEvent**() y cuatro métodos más definidos para realizar el proceso correspondiente a cada uno de los cuatro comandos generados por las opciones del menú.

Revisando el cuerpo de **handleEvent**() hay que tener dos cosas en cuenta: no debemos procesar el contenido del campo **message.command** si antes no hemos comprobado, por medio del campo **what**, que lo que se ha recibido es un comando. El borrado de un evento, por medio de **clearEvent**(), sólo se debe efectuar cuando nosotros hayamos procesado el evento.

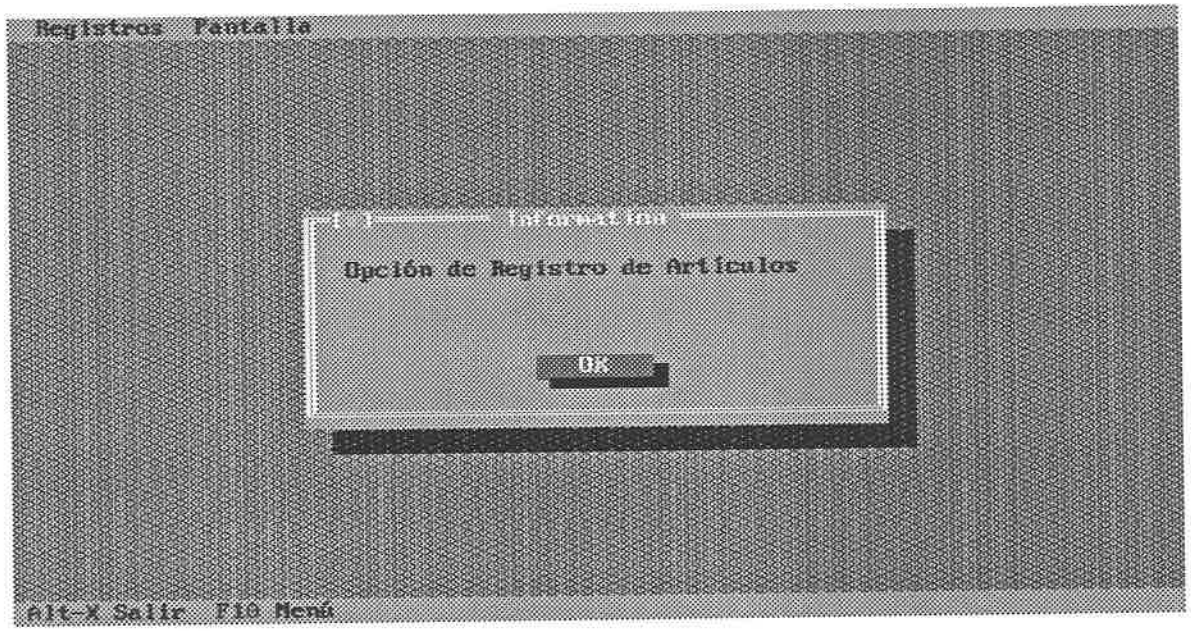
Como verá he usado la función **messageBox**() para que los métodos correspondientes a las opciones del menú hagan aparecer un mensaje en pantalla. Esa es ahora mismo su única funcionalidad, pero es suficiente para comprobar que al seleccionar una determinada opción se ejecuta el método adecuado.

De forma similar a como hemos procesado los comandos generados por las opciones del menú, podríamos añadir nuevos comandos a la línea de estado y definir los métodos correspondientes, añadiendo en el método **handleEvent**() las líneas necesarias para su proceso.

```
//
// SEMTV04.CPP                  Gestión de los comandos del menú
//

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox // Para usar la función global messageBox()

#include <Tv.H>         // Se incluye el archivo de cabecera Turbo Vision

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )

                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;

} ;

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu )
                              ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla

        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                             * new TSubMenu( "~R~egistros", kbAltR ) +

                                 * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                                 * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                                 newLine() +
                                 * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +
```

```
            * new TSubMenu( "~P~antalla", kbAltP ) +

                * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                        ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )      // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;


                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :

                                mConfigurarColores() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ;// Regresa sin procesarlo
                }

                clearEvent( Evento ) ;  // Si se ha procesado hay que borrarlo
        }
}
```

```
        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
        messageBox( "Opción de Registro de Libros", mfOKButton | mfInformation ) ;
        }

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```


Registros Pantalla
Información
Opción de Registro de Artículos
OK
Alt-X Salir F10 Menú

SEMTV04

CAJAS DE DIALOGO

Las cajas de diálogo son la herramienta básica para la introducción de datos en una aplicación Turbo Vision. Una caja de diálogo es una ventana en la que se insertan distintos elementos, como pueden ser líneas de edición, cajas de selección de opciones múltiples o exclusivas y botones.

Cajas de diálogo modales y no modales

Para presentar una caja de diálogo en pantalla tendremos que crear un objeto **TDialog**, insertar en él los controles que deseemos, botones, cajas de selección, etc., y a continuación podemos ejecutarla o insertarla en el despacho.

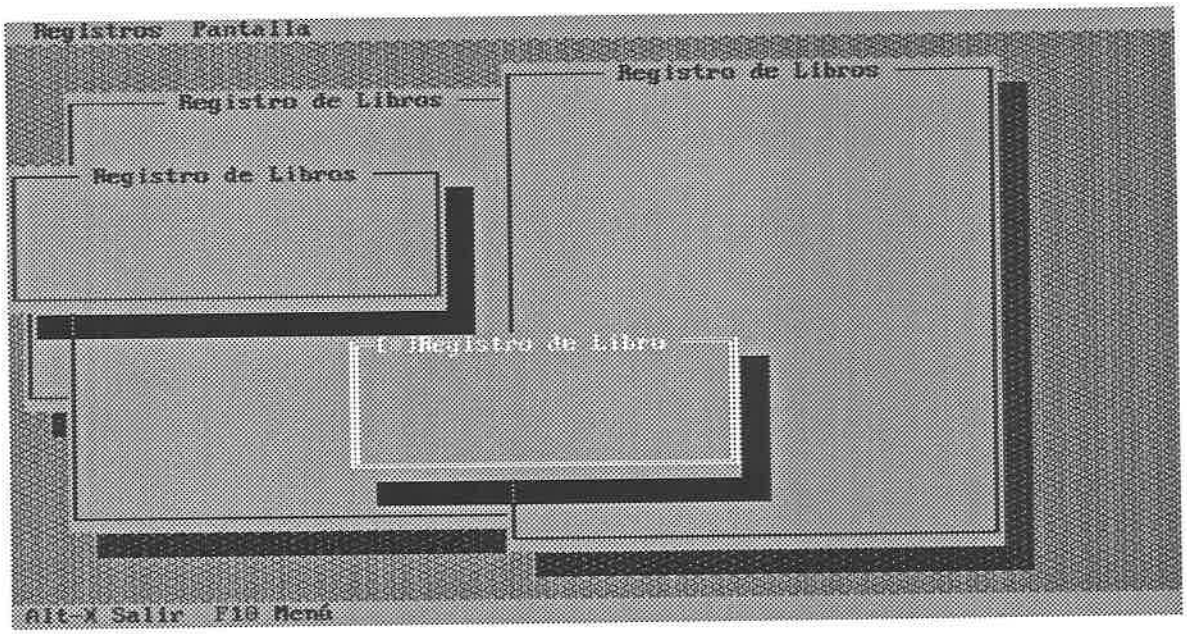
El constructor de la clase **TDialog** necesita como parámetros un objeto **TRect**, definiendo las esquinas superior izquierda e inferior derecha de la caja, y una cadena especificando el título. Una caja así, sin tener ningún elemento insertado, aparece como una ventana con un sólo control, el icono de cierre.

Teniendo el objeto creado tan sólo hemos de insertarlo en el despacho, para lo que utilizaremos el método **insert()** sobre el objeto **deskTop**, que hemos heredado de **TApplication** y representa el despacho. En el listado SEMTV05.CPP puede ver como se ha modificado el método **mRegistroLibro** para que defina una caja de diálogo con una posición y dimensiones aleatorias y la inserte.

```
//
// SEMTV05.CPP                    Se añade la creación de una caja de diálogo
//

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
```

```
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                & initMenuBar,
                                & initDeskTop )

                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;

} ;

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu )
                              ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla

        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                        * new TSubMenu( "~R~egistros", kbAltR ) +

                                * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                                * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                                newLine() +
                                * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

                        * new TSubMenu( "~P~antalla", kbAltP ) +

                                * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                                * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                      ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;
```

```
                case cmRegistroArticulo :

                        mRegistroArticulo() ;
                        break ;

                case cmConfigurarModo :

                        mConfigurarModo() ;
                        break ;

                case cmConfigurarColores :

                        mConfigurarColores() ;
                        break ;

                default :   // Si no es ninguno de los comandos reconocidos

                        return ; // Regresa sin procesarlo
            }

            clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
            // Se define una caja de diálogo en una posición y con un
            // tamaño aleatorios

        TDialog * RegistroLibro = new TDialog( TRect( random(10), random(10),
                                                      random(30)+10, random(15)+10 ),
                                                      "Registro de Libros" ) ;

        deskTop->insert( RegistroLibro ) ;      // Se inserta en el despacho
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```



SEMTV05

Al ejecutar el programa y seleccionar la opción correspondiente verá como se abre una ventana. Si repite el proceso aparecerá otra, y otra. El hecho de que mientras la caja de diálogo está abierta en el despacho sea posible acceder al menú significa que esa caja no es modal, ya que si lo fuese todos los eventos serían procesados por ella, impidiéndose el acceso a cualquier otra vista hasta en tanto no se cierre.

Para convertir la caja en una caja de diálogo modal, tan sólo hemos de cambiar la llamada al método **insert**() por una llamada a **execView**. La diferencia, al ejecutar, es la que he comentado antes, no es posible acceder ninguna otra vista sin cerrar previamente la caja. En el listado SEMTV06.CPP puede ver esta versión del programa. Fíjese en que también he añadido una llamada al método **destroy**() para liberar la memoria asignada al crear el objeto **TDialog**.

En principio esta caja de diálogo, que no tiene control alguno, se puede cerrar de dos formas: pulsando el icono de cierre con el puntero del ratón, o bien pulsar la tecla ESC. Sin embargo existen dos métodos más a los que podemos estar acostumbrados, la

combinación Alt-F3, usado en los IDE's de Borland, y la pulsación de Ctrl-F4, que sirve para cerrar una ventana en el entorno Windows. Para añadir esta funcionalidad lo único que tenemos que hacer es asociar estas pulsaciones de tecla con el comando **cmClose**, sirviéndonos de los campos de la línea de estado. También puede ver esto en el mismo listado.

```
        //
        // SEMTV06.CPP                  Crea una caja de diálogo modal
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
 {
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )
                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;

} ;

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
 {
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                                * new TStatusItem( 0, kbAltF3, cmClose ) +
                                * new TStatusItem( 0, kbCtrlF4, cmClose )
                              ) ;
```

```
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla

        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
           * new TSubMenu( "~R~egistros", kbAltR ) +

                * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                newLine() +
                * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

           * new TSubMenu( "~P~antalla", kbAltP ) +

                * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                        ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;

                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :

                                mConfigurarColores() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
                // Se define una caja de diálogo

        TDialog * RegistroLibro = new TDialog( TRect( 10, 10, 50, 20 ),
                                                "Registro de Libros" ) ;

        deskTop->execView( RegistroLibro ) ;// Se ejecuta

        destroy( RegistroLibro ) ;
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores
```

```
void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}
```

```
int main()
 {
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
 }
```

Inserción de controles

Desde luego una caja de diálogo vacía no es muy útil. Para darle una mayor funcionalidad tendremos que insertar en ella controles, que no son mas que vistas especializadas en la entrada de datos. Existen distintos tipos de controles, dependiendo de la estructura de la información que deseamos capturar. Vamos a ir viendo en detalle cada uno de ellos.

Puesto que vamos a registrar libros, lo primero que desearemos tener es el título, autor y editorial. Estos tres campos tienen algo en común, y es que son secuencias de caracteres que tendremos que teclear, ya que no es posible dar un conjunto de títulos entre los que elegir simplemente marcando, porque sería una lista interminable. Para crear un campo de este tipo nos serviremos de la clase **TInputLine**, cuyo constructor necesita como parámetros un objeto **TRect**, definiendo unas coordenadas para el campo, y un entero indicando la longitud máxima. Una vez creado el objeto lo insertamos en la caja de diálogo, usando el mismo método **insert()** que utilizamos para insertar objetos en el despacho, al fin y al cabo tanto el despacho como una caja de diálogo son vistas, y comparten muchos miembros.

Si disponemos tres objetos **TInputLine** en la cada de diálogo, el usuario difícilmente sabrá qué es lo que tiene que introducir en cada uno de ellos, por lo que tendremos que añadir un título o etiqueta a cada uno de ellos. Con este fin crearemos un objeto **TLabel** por cada línea de entrada, especificando las coordenadas donde ha de mostrarse, el título y el objeto **TInputLine** al que está asociado. Este último vínculo nos servirá para desplazarnos a una determinada línea de entrada pulsando con el puntero del ratón en su título, es decir, no se trata de un texto inerte en la pantalla, sino que es sensible a eventos, como otra vista más, aunque no genere información alguna.

No tiene mucho sentido, como he comentado anteriormente, preparar un conjunto de títulos, y tampoco de autores, ya que sería difícil recopilar todos los títulos posibles, y dado que no es normal que tengamos dos veces el mismo título, no tiene mucho sentido. Sin embargo en el caso de la editorial esto cambia, ya que no hay tantas, y además los libros que podemos tener en casa seguramente se reducen a unas cuantas. Basándonos en esto, sí sería útil que la línea correspondiente a la editorial tuviese una lista histórica, con el fin de poder entrar una editorial simplemente seleccionándola, siempre y cuando la hayamos insertado alguna vez antes, ya que la lista histórica se va alimentando de nuestras propias entradas.

Una lista histórica va por lo tanto asociada al objeto **TInputLine** en la que deseamos que aparezca. Para crearla nos serviremos de la clase **THistory**, cuyo constructor toma como parámetros las coordenadas donde aparecerá el indicador de lista histórica, una flecha hacia abajo, la línea de entrada a la que está asociada y un identificador, que servirá para tener codificadas las distintas líneas históricas que podemos tener en distintos campos, de tal forma que a igual identificador misma lista, es decir, dos líneas de entrada distintas pueden compartir la misma lista histórica.

En el listado SEMTV07.CPP puede ver como se han añadido las tres líneas de entrada con sus tres etiquetas y la lista histórica. Al ejecutarlo la lista histórica aparece vacía, pero a medida que vayas tecleando en la línea distintos valores estos irán almacenándose en ella.

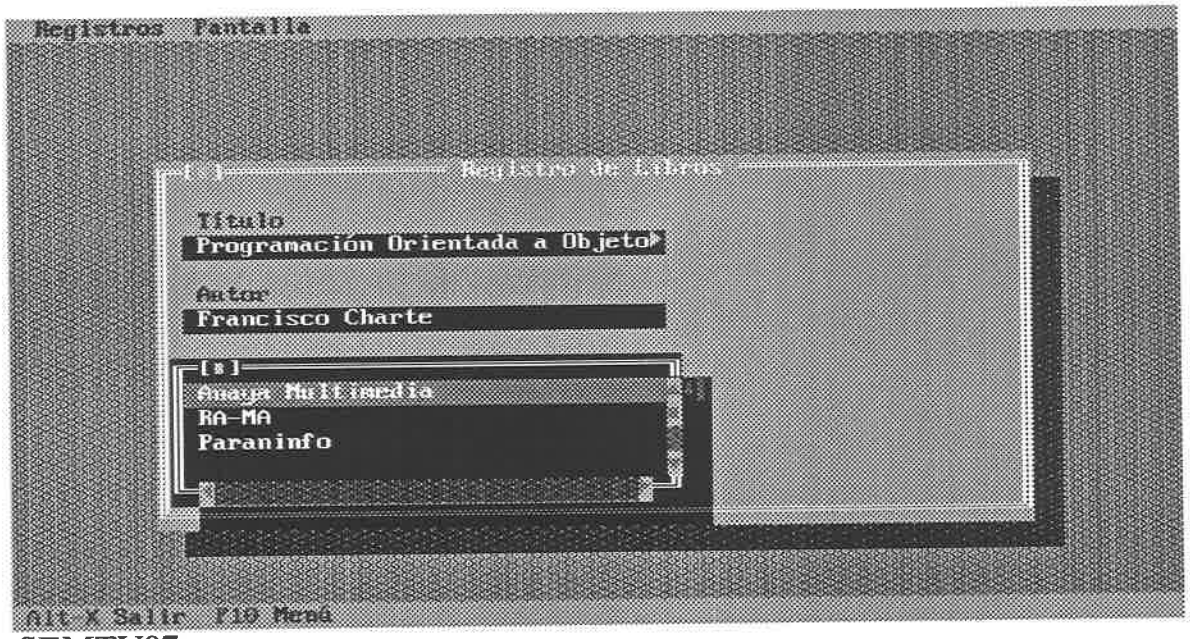
Algo más que podemos desear saber acerca del libro es su tema, puede ser una novela, un libro de poesía o de informática, y otras opciones, como el estar ilustrado, contener un disco o ser una colección. Veamos el primer campo, el tema. Un libro puede ser o una novela, o un libro de poesía o un libro de informática, pero sólo uno de ellos al mismo tiempo, es decir, el seleccionar una de las opciones excluye a las otras dos. Para esto existen las cajas de selección exclusiva, también conocidas como botones de radio, que son objetos de la clase **TRadioButtons**. El segundo caso sin embargo no es así, ya que un libro puede ser ilustrado o no, idenpendientemente de que traiga o no un disco y de que pertenezca o no a una colección. En este caso tenemos una caja de selección múltiple, que son objetos de la clase **TCheckBoxes**.

Conociendo esta diferencia fundamental, un objeto **TRadioButtons** y un objeto **TCheckBoxes** tienen mucho en común, ya que de hecho derivan de una misma clase. Su constructor, tanto el de una como el de otra, necesita un objeto **TRect** que define el área donde se visualizará la caja de selección, y una lista de objetos **TSItem** definiendo las distintas opciones que la componen.

```
        //
        // SEMTV07.CPP                  Se añaden líneas de entrada de datos a la caja
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory

#include <Tv.H>         // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
 {
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )

                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;

} ;

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                                * new TStatusItem( 0, kbAltF3, cmClose ) +
                                * new TStatusItem( 0, kbCtrlF4, cmClose )
                              ) ;
}

        // Implementación del método initStatusBar


TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
 {
                // Se define como área de menús la primera línea de pantalla
```

```
        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
            * new TSubMenu( "~R~egistros", kbAltR ) +

                * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                newLine() +
                * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

            * new TSubMenu( "~P~antalla", kbAltP ) +

                * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                        ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;


                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :

                                mConfigurarColores() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;    // Si se ha procesado hay que borrarlo
        }
        }

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
                // Se define una caja de diálogo

        TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                                "Registro de Libros" ) ;

        TInputLine * Linea ; // Un puntero a un objeto TInputLine
        TLabel * Etiqueta ; // Etiqueta de la línea

                // Se crea una línea con una entrada de 50 caracteres como máximo

        Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 50 ) ;

                // Se define una etiqueta enlazado al campo anterior

        Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

        RegistroLibro -> insert( Linea ) ;      // Se insertan el campo y la
        RegistroLibro -> insert( Etiqueta ) ;   // etiqueta en la caja

                // Se repite el proceso para los otros dos campos

        Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 50 ) ;
        Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;

        RegistroLibro -> insert( Linea ) ;
        RegistroLibro -> insert( Etiqueta ) ;

        Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 50 ) ;
        Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

        RegistroLibro -> insert( Linea ) ;
        RegistroLibro -> insert( Etiqueta ) ;
```

```
        // Se crea una lista histórica de editoriales
        RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;
        deskTop->execView( RegistroLibro ) ;    // Se ejecuta
        destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}
        // Implementación del método mRegistroArticulo
void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}
        // Implementación del método mConfigurarModo
void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}
        // Implementación del método mConfigurarColores
void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}
int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto
        MiAplicacion . run() ; // Y se ejecuta
        return 0 ; // Fin del programa
}
```



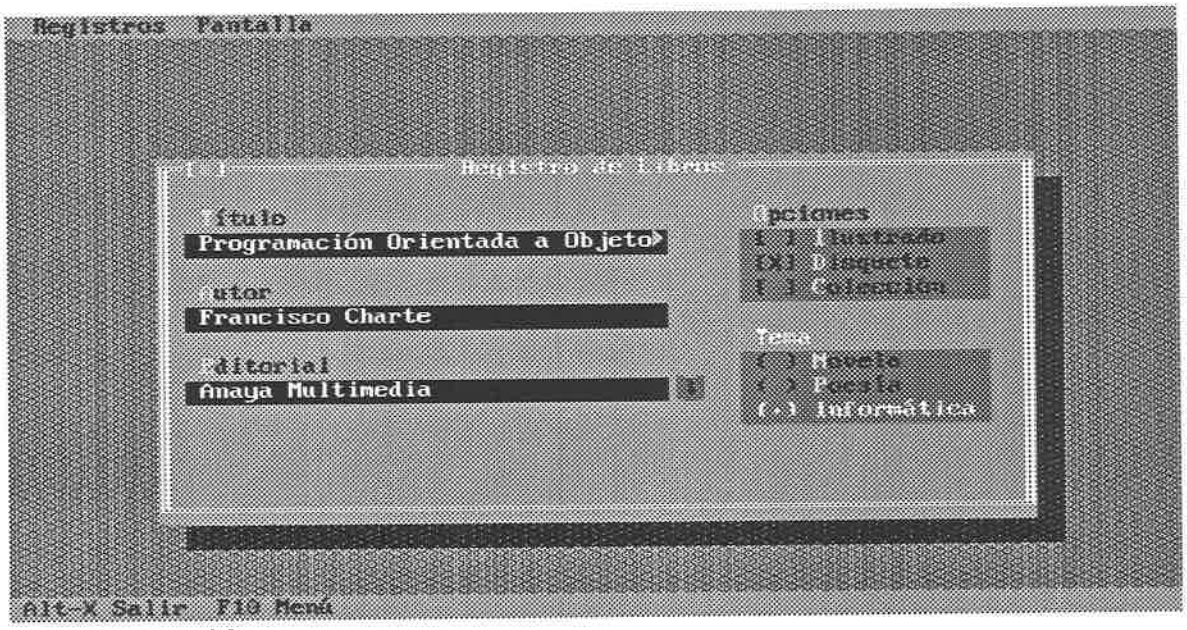
SEMTV07

Al igual que a las líneas de entrada de datos, a las cajas de selección se puede asignar una etiqueta creando un objeto **TLabel** asociado. En el listado SEMTV08.CPP puede ver como se han añadido a la caja de diálogo las dos cajas de selección comentadas.

```
        //
        // SEMTV08.CPP                  Se añaden cajas de selección múltiples y exclusivas
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory
#define Uses_TCheckBoxes
#define Uses_TRadioButtons
#define Uses_TSItem

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
 {
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )

                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;

} ;

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
```

```
                        * new TStatusItem( 0, kbAltF3, cmClose ) +
                        * new TStatusItem( 0, kbCtrlF4, cmClose )
                    ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla

        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
           * new TSubMenu( "~R~egistros", kbAltR ) +

                * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                newLine() +
                * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

           * new TSubMenu( "~P~antalla", kbAltP ) +

                * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                        ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;

                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :

                                mConfigurarColores() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;    // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
                // Se define una caja de diálogo

        TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                                "Registro de Libros" ) ;

        TInputLine * Linea ; // Un puntero a un objeto TInputLine
        TLabel * Etiqueta ; // Etiqueta de la línea

                // Se crea una línea con una entrada de 50 caracteres como máximo

        Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 50 ) ;

                // Se define una etiqueta enlazado al campo anterior

        Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

        RegistroLibro -> insert( Linea ) ; // Se insertan el campo y la
        RegistroLibro -> insert( Etiqueta ) ;       // etiqueta en la caja

                // Se repite el proceso para los otros dos campos

        Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 50 ) ;
        Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;
```

```
RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 50 ) ;
Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una lista histórica de editoriales

RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;

        // Se define una caja de selecciones múltiples con
        // tres posibles campos

TCheckBoxes * Opciones = new TCheckBoxes( TRect( 40, 3, 57, 6 ),
                                  new TSItem( "~I~lustrado",
                                  new TSItem( "~D~isquete",
                                  new TSItem( "~C~olección",
                                  0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 2, 55, 3 ), "~O~pciones", Opciones ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Opciones ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una caja de selección exclusiva con
        // tres posibles opciones

TRadioButtons * Tema = new TRadioButtons( TRect( 40, 8, 57, 11 ),
                                  new TSItem( "~N~ovela",
                                  new TSItem( "~P~oesía",
                                  new TSItem( "In~f~ormática",
                                  0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 7, 55, 8 ), "Te~m~a", Tema ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Tema ) ;
RegistroLibro -> insert( Etiqueta ) ;

        deskTop->execView( RegistroLibro ) ;    // Se ejecuta

        destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```



SEMTV08

Una vez hayamos introducido toda la información solicitada en los campos anteriores, tendremos que indicar si estamos de acuerdo y queremos almacenarlos, si queremos cancelar, si queremos buscar algún registro, etc. Para esto están los botones, controles que al pulsarse causan la salida de la ejecución de la caja de diálogo, devolviendo un código que nuestro método puede utilizar para proceder de una u otra forma.

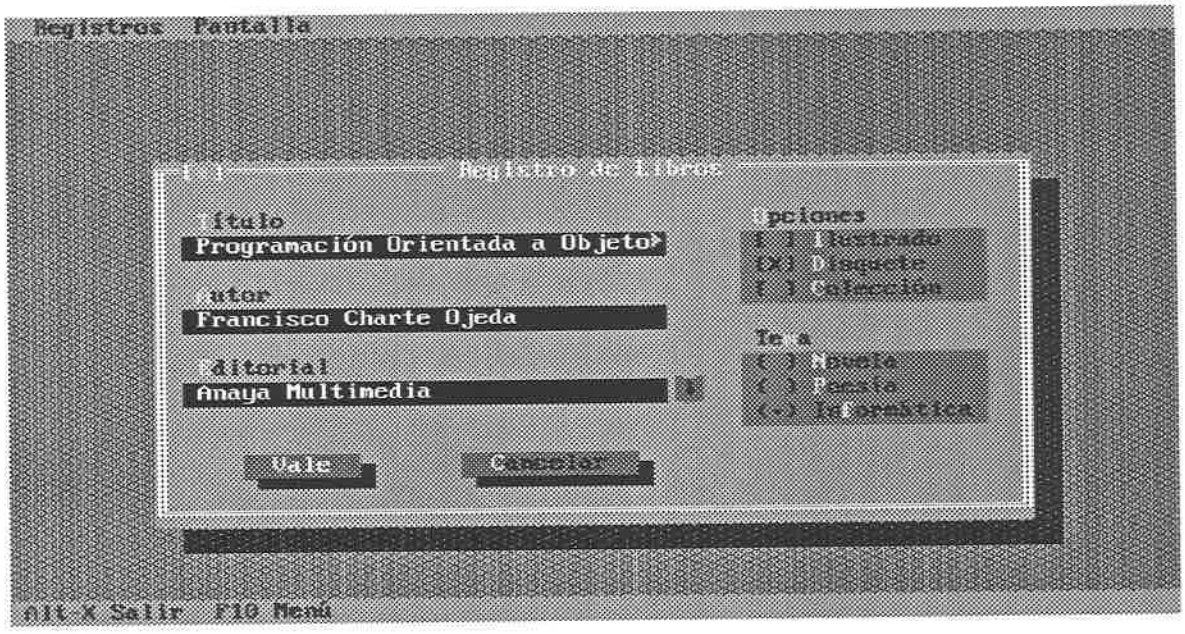
Para definir un botón crearemos un objeto de la clase **TButton**, facilitando el área donde se visualizará, por medio de un objeto **TRect**, el texto que debe aparecer dentro del botón, el comando o código que debe generar el botón y el tipo de botón. El comando a generar puede ser uno de los ya existentes, como **cmOK** o **cmCancel**, o uno creado por nosotros mismos. El tipo del botón puede ser **bfNormal** o **bfDefault**, indicando que es un botón normal o el botón a usar por defecto, respectivamente.

En el listado SEMTV09.CPP puede ver cómo se han añadido dos botones a la caja de diálogo, y cómo se recogen el comando del botón pulsado. Al compilar el programa obtendrá un aviso, ya que el valor recogido en la variable **Boton**, no se utiliza.

```
        //
        // SEMTV09.CPP                  Se añaden botones a la caja de diálogo
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory
#define Uses_TCheckBoxes
#define Uses_TRadioButtons
#define Uses_TSItem
#define Uses_TButton

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )

                        {}

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;

} ;

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                                * new TStatusItem( 0, kbAltF3, cmClose ) +
                                * new TStatusItem( 0, kbCtrlF4, cmClose )
                              ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
```

```
{
        // Se define como área de menús la primera línea de pantalla

    Area . b . y = Area . a . y + 1 ;

    return new TMenuBar( Area,
      * new TSubMenu( "~R~egistros", kbAltR ) +

        * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
        * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
        newLine() +
        * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

      * new TSubMenu( "~P~antalla", kbAltP ) +

        * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
        * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
            ) ;
}

    // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
        // Primero se ha de procesar el evento por parte
        // del gestor de eventos de nuestra clase base

    TApplication::handleEvent( Evento ) ;

    if( Evento . what == evCommand )        // Si el evento es un comando
    {
        switch( Evento . message . command ) // Según el comando que sea
        {
            case cmRegistroLibro :

                mRegistroLibro() ;      // Llama al método adecuado
                break ;

            case cmRegistroArticulo :

                mRegistroArticulo() ;
                break ;

            case cmConfigurarModo :

                mConfigurarModo() ;
                break ;

            case cmConfigurarColores :

                mConfigurarColores() ;
                break ;

            default :   // Si no es ninguno de los comandos reconocidos

                return ; // Regresa sin procesarlo
        }

        clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
    }
}

    // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
        // Se define una caja de diálogo

    TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                           "Registro de Libros" ) ;

    TInputLine * Linea ; // Un puntero a un objeto TInputLine
    TLabel * Etiqueta ; // Etiqueta de la línea

        // Se crea una línea con una entrada de 50 caracteres como máximo

    Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 50 ) ;

        // Se define una etiqueta enlazado al campo anterior

    Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

    RegistroLibro -> insert( Linea ) ; // Se insertan el campo y la
    RegistroLibro -> insert( Etiqueta ) ;       // etiqueta en la caja

        // Se repite el proceso para los otros dos campos

    Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 50 ) ;
    Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;

    RegistroLibro -> insert( Linea ) ;
    RegistroLibro -> insert( Etiqueta ) ;

    Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 50 ) ;
    Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

    RegistroLibro -> insert( Linea ) ;
    RegistroLibro -> insert( Etiqueta ) ;
```

```
        // Se crea una lista histórica de editoriales

RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;

        // Se define una caja de selecciones múltiples con
        // tres posibles campos

TCheckBoxes * Opciones = new TCheckBoxes( TRect( 40, 3, 57, 6 ),
                                new TSItem( "~I~lustrado",
                                new TSItem( "~D~isquete",
                                new TSItem( "~C~olección",
                                0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 2, 55, 3 ), "~O~pciones", Opciones ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Opciones ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una caja de selección exclusiva con
        // tres posibles opciones

TRadioButtons * Tema = new TRadioButtons( TRect( 40, 8, 57, 11 ),
                                new TSItem( "~N~ovela",
                                new TSItem( "~P~oesía",
                                new TSItem( "In~f~ormática",
                                0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 7, 55, 8 ), "Te~m~a", Tema ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Tema ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se insertan un botón Vale y un botón Cancelar

RegistroLibro -> insert( new TButton( TRect( 5, 12, 15, 14 ), "~V~ale", cmOK, bfDefault )) ;
RegistroLibro -> insert( new TButton( TRect( 20, 12, 34, 14 ), "~C~ancelar", cmCancel, bfNormal )) ;

        // Se ejecuta la caja de diálogo obteniendo el código
        // del botón pulsado

        ushort Boton = deskTop->execView( RegistroLibro ) ;

        destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```



SEMTV09

Asignación y obtención de datos

Al igual que no era muy útil una caja de diálogo vacía, tampoco tiene mucho sentido el capturar datos para luego no procesarlos en algún modo, como pueda ser almacenándolos en un archivo. También es probable que en principio los campos que forman la caja deban tener un valor inicial, en lugar de estar vacíos.

Como hemos visto en los ejemplos anteriores, básicamente tenemos dos tipos de datos: una línea de caracteres, que no es mas que una matriz del tipo **char**, y cajas de selección, múltiples o exclusivas, que se representan por un entero sin signo, un tipo **ushort**. En este tipo de dato es posible almacenar un máximo de 16 opciones en una selección múltiple, representándose el estado de cada una de ellas en cada momento por el bit correspondiente, sabiendo que el bit 0, el primero por la derecha, representa la primera de las opciones. En una selección exclusiva, por el contrario, al no poder estar seleccionada más de una opción este tipo de dato es capaz de representar hasta 65536 opciones, sabiendo que el 0 corresponde a la primera de ellas.

Sabiendo esto es fácil construcir una estructura donde almacenar los campos de la caja de diálogo. Una vez creada podemos dar valores iniciales, y fijarlos en los controles de la caja con el método **setData()** de **TDialog**, al que facilitaremos la dirección donde se encuentra la información, es decir, un puntero a la estructura que hemos creado. De igual forma podemos obtener los valores que actualmente tienen los controles de la caja, almacenándolos en nuestra estructura, utilizando el método **getData()**, que necesita el mismo parámetro.

Hay que tener un especial cuidado en que el tamaño de la estructura que preparamos para el almacenamiento de los datos coincida con el de los controles definidos en la caja de diálogo, ya que Turbo Vision no lo comprueba en modo alguno. Nosotros si lo podemos hacer, usando el operador **sizeof()** para obtener el tamaño de la estructura y el método **dataSize()** para conocer el de los controles insertados. El valor devuelto por ambos debe coincidir.

En el listado SEMTV10.CPP puede ver un ejemplo del uso de los métodos comentados anteriormente. Observe cómo la inicialización de la estructura **Libro** se efectúa una sola vez, en el constructor de la aplicación, de tal forma que si llamamos sucesivas veces a la caja de diálogo nos encontraremos con los últimos valores que hayamos introducido.

```
//
// SEMTV10.CPP                Se recogen los datos entrados en la caja
//

#define Uses_TApplication      // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory
#define Uses_TCheckBoxes
#define Uses_TRadioButtons
#define Uses_TSItem
#define Uses_TButton
```

```
#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

#include <String.H> // Para usar la función strcpy()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203 ;

        // Se define una estructura que nos sirva
        // para almacenar la información de la caja de diálogo

struct {
        char Titulo[51] ;
        char Autor[51] ;
        char Editorial[51] ;
        ushort Opciones ;
        ushort Tema ;
} Libro ;
```

```
//
// Se deriva una clase a partir de TApplication,
// a la que se llama AplicacionEstandar
//

class AplicacionEstandar : public TApplication
{
    public :

        // Se define el constructor de la clase

        AplicacionEstandar() ;

        // Método para inicializar la línea de estado

        static TStatusLine * initStatusLine( TRect ) ;

        // Método para inicializar la barra de menú

        static TMenuBar * initMenuBar( TRect ) ;

        // Método para la gestión de eventos propios

        virtual void handleEvent( TEvent & ) ;

        // Estos son los métodos miembros a los que
        // se llamará dependiendo del comando recibido

        void mRegistroLibro() ;
        void mRegistroArticulo() ;
        void mConfigurarModo() ;
        void mConfigurarColores() ;

};

// Implementación del constructor

AplicacionEstandar::AplicacionEstandar() :

        TProgInit( & initStatusLine,
                   & initMenuBar,
                   & initDeskTop )

{
    // Se inicializan los miembros de la estructura

    strcpy( Libro . Titulo, "" ) ;
    strcpy( Libro . Autor, "" ) ;
    strcpy( Libro . Editorial, "" ) ;

    Libro . Opciones = Libro . Tema = 0 ;
}

// Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
    // Se define como área de línea de estado la última línea
    // del área total

    Area . a . y = Area . b . y - 1 ;

    // Se crea una línea de estado y se devuelve

    return new TStatusLine( Area,
                * new TStatusDef( 0, 65535 ) +
                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                * new TStatusItem( 0, kbAltF3, cmClose ) +
                * new TStatusItem( 0, kbCtrlF4, cmClose )
              ) ;
}

// Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
    // Se define como área de menús la primera línea de pantalla

    Area . b . y = Area . a . y + 1 ;

    return new TMenuBar( Area,
                * new TSubMenu( "~R~egistros", kbAltR ) +

                    * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                    * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                    newLine() +
                    * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

                * new TSubMenu( "~P~antalla", kbAltP ) +

                    * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                    * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
              ) ;
```

```
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;

                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :

                                mConfigurarColores() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
                // Se define una caja de diálogo

TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                        "Registro de Libros" ) ;

TInputLine * Linea ; // Un puntero a un objeto TInputLine
TLabel * Etiqueta ; // Etiqueta de la línea

        // Se crea una línea con una entrada de 50 caracteres como máximo

Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 51 ) ;

        // Se define una etiqueta enlazado al campo anterior

Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

RegistroLibro -> insert( Linea ) ; // Se insertan el campo y la
RegistroLibro -> insert( Etiqueta ) ;       // etiqueta en la caja

        // Se repite el proceso para los otros dos campos

Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 51 ) ;
Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;

RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 51 ) ;
Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una lista histórica de editoriales

RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;

        // Se define una caja de selecciones múltiples con
        // tres posibles campos

TCheckBoxes * Opciones = new TCheckBoxes( TRect( 40, 3, 57, 6 ),
                                new TSItem( "~I~lustrado",
                                new TSItem( "~D~isquete",
                                new TSItem( "~C~olección",
                                0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 2, 55, 3 ), "~O~pciones", Opciones ) ;
```

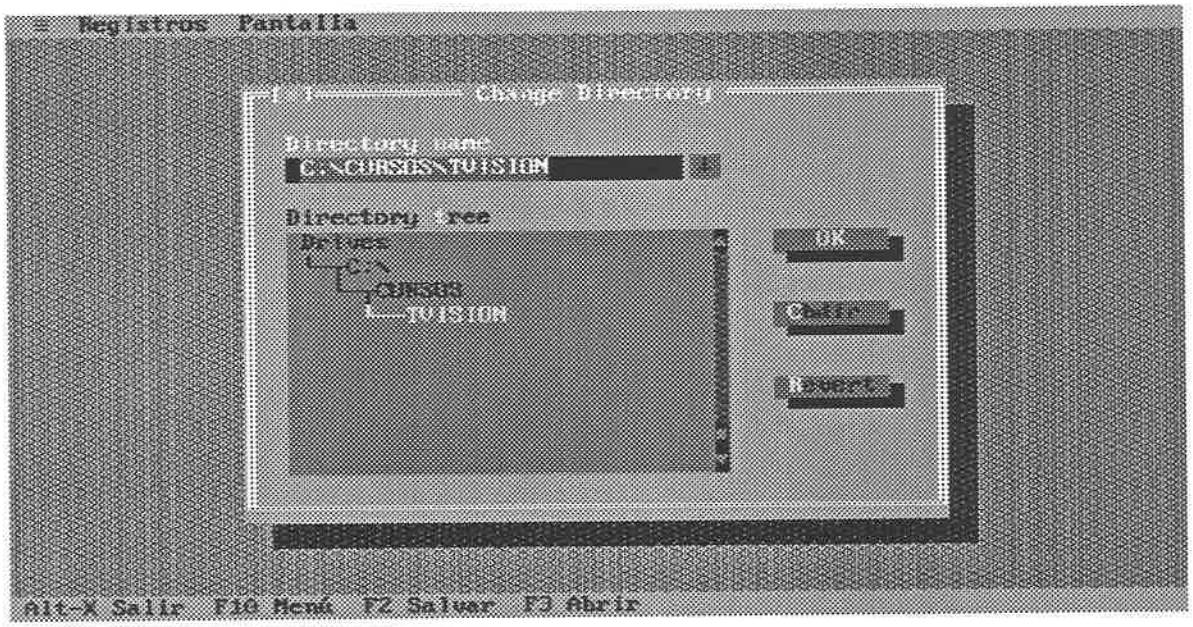
```
        // Se insertan ambos controles

RegistroLibro -> insert( Opciones ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una caja de selección exclusiva con
        // tres posibles opciones

TRadioButtons * Tema = new TRadioButtons( TRect( 40, 8, 57, 11 ),
                                new TSItem( "~N~ovela",
                                new TSItem( "~P~oesía",
                                new TSItem( "In~f~ormática",
                                0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 7, 55, 8 ), "Te~m~a", Tema ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Tema ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se insertan un botón Vale y un botón Cancelar

RegistroLibro -> insert( new TButton( TRect( 5, 12, 15, 14 ), "~V~ale", cmOK, bfDefault )) ;
RegistroLibro -> insert( new TButton( TRect( 20, 12, 34, 14 ), "~C~ancelar", cmCancel, bfNormal )) ;

RegistroLibro -> setData( & Libro ) ; // Se fijan los valores de los campos

        // Se comprueba que los controles insertandos coincidan
        // en tamaño con los campos definidos en la estructura Libro

if( sizeof( Libro ) != RegistroLibro -> dataSize() )

        messageBox( "El tamaño de la estructura no coincide", mfOKButton | mfWarning ) ;

        // Se ejecuta la caja de diálogo obteniendo el código
        // del botón pulsado

ushort Boton = deskTop->execView( RegistroLibro ) ;

if( Boton == cmOK )     // Si se aceptó el registro de este libro
{
                // Se recoge la información

        RegistroLibro -> getData( & Libro ) ;

                // A continuación se puede realizar cualquier
                // proceso con la información, como su almacenamiento
                // en un archivo
        }

        destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
{
        messageBox( "Opción de Configuración de Colores", mfOKButton | mfInformation ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```

Cajas de diálogo estandar

Aunque en la mayoría de las ocasiones tendremos que diseñar cajas de diálogo a medida para cada aplicación, en otras nos encontraremos con cajas exactamente iguales, cajas que además, existen en otras aplicaciones como el propio IDE de Borland. A éstas se les llama cajas de diálogo estándar, y básicamente son tres: una caja de selección de directorio o cambio de directorio, una de selección de archivo y otra de selección de colores. Además veremos una forma estándar de editar el contenido de un archivo, gracias a una clase de Turbo Vision que integra un potente editor.

Si desde el interior de nuestra aplicación vamos a permitir cambiar el directorio actual, tendremos que preparar una caja de diálogo un tanto especial, donde exista una lista con los directorios existentes y sea posible ir desplazándose por ellos, hasta seleccionar el deseado. Preparar esta caja de diálogo puede ser algo más complicado, a no ser que nos sirvamos de la clase **TChDirDialog**. Al crear un objeto de esta clase tendremos que facilitar dos parámetros: un entero conteniendo opciones que modifiquen el comportamiento de la caja, y que puede ser **cdNormal** para generar una caja normal y de uso inmediato, **cdNoLoadDir** si no deseamos que se lea automáticamente el directorio para fijarlo después nosotros y **cdHelpButton** si queremos incluir en la caja un botón de ayuda; el segundo parámetro también es un entero, especificando la lista histórica que usará la caja para recordar selecciones anteriores de directorios, sabiendo que si dos cajas de diálogo tienen el mismo identificador compartirán la misma lista histórica. Con esto ya tendremos un objeto **TChDirDialog**, que tan sólo tendremos que ejecutar para conseguir el cambio de directorio.

Otra operación común en una aplicación es la selección de un archivo, independientemente de la finalidad que se le de. La preparación de una caja de diálogo para ello es tan simple como en el caso anterior, ya que la clase **TFileDialog** se ocupa de todo. Al crear un objeto de esta clase tendremos que pasar los parámetros: una cadena con la referencia de selección de archivos, conteniendo los habituales caracteres comodines; un título para la caja de diálogo; un título para la línea de entrada del nombre; un entero especificando qué botones se han de incluir en la caja y por último un identificador de lista histórica. Los posibles botones a incluir en una caja de diálogo de selección de archivo son **fdOKButton**, **fdOpenButton**, **fdReplaceButton**, **fdClearButton** y **fdHelpButton**. Al ejecutar la caja de diálogo obtendremos devuelto el valor **cmFileOpen**, **cmFileReplace** o **cmFileClear**, dependiendo del botón que se haya pulsado.

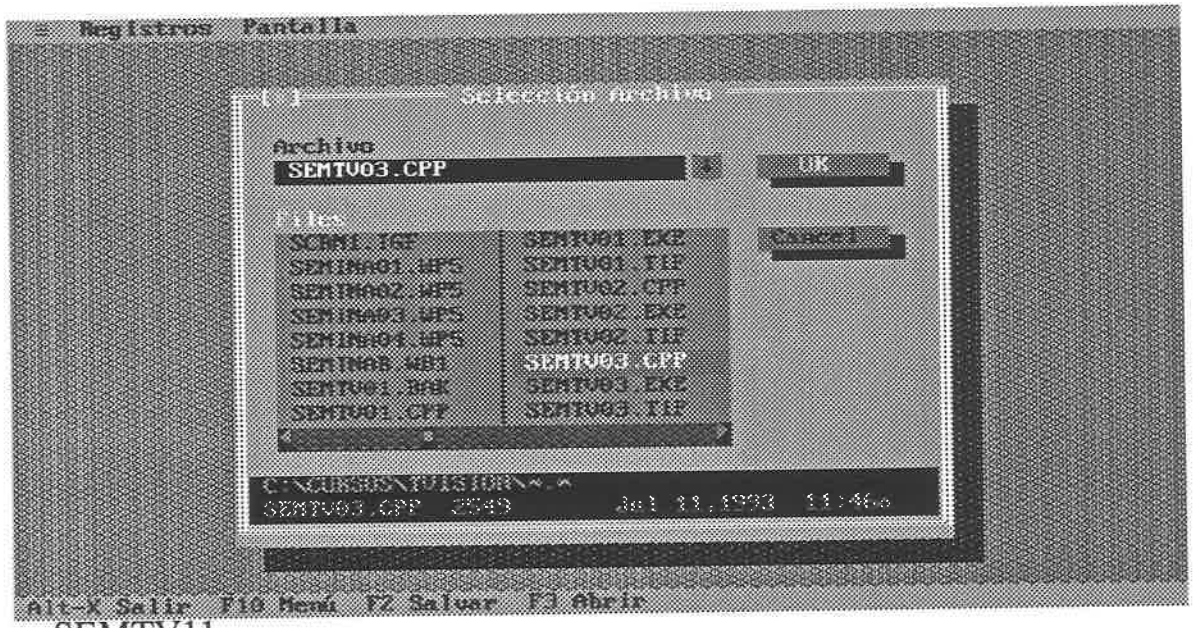


SEMTV11

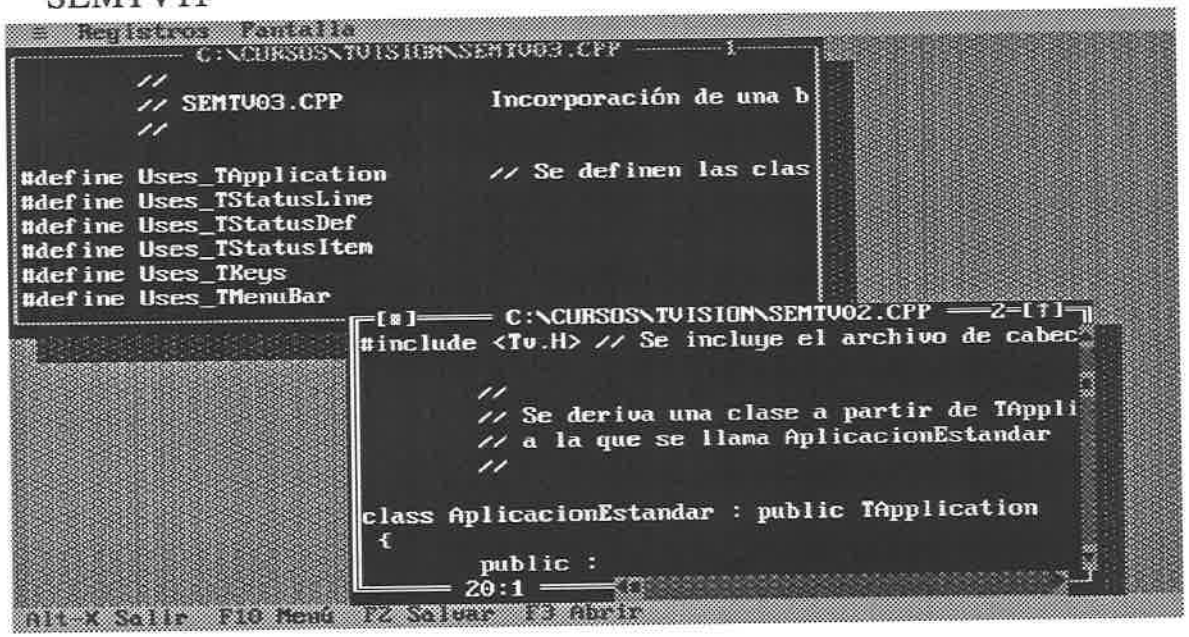
Una vez ejecutada la caja de diálogo podremos obtener el nombre del archivo que se ha seleccionado utilizando el método **getFileName**(), al que facilitaremos como parámetro una cadena donde nos será devuelto el campo, que podrá tener hasta **MAXPATH** caracteres.

Ya que tenemos la posibilidad de que el usuario seleccione un archivo, por qué no permitir que lo edite. Turbo Vision incorpora varias clases que definen un editor, capaz de manipular el contenido de un archivo o editar un campo amplio en una caja de diálogo. La clase **TEditWindow** recogen el contenido de un archivo, lo visualiza en una ventana y permite su edición. Para ello tan sólo tendremos que facilitar la extensión en la que se ha de visualizar la ventana, un objeto **TRect**, el nombre del archivo y el número de la ventana. Creado el objeto bastará con insertarlo en el despacho, teniendo en cuenta además que habrá que activar los comandos que deseemos, apertura, grabación, etc., en la línea de estado.

Por último nos encontramos con la caja de diálogo que nos permite seleccionar los colores con los que deseamos visualizar las distintas vistas de nuestra aplicación, creada por medio de un objeto **TColorDialog**.



SEMTV11

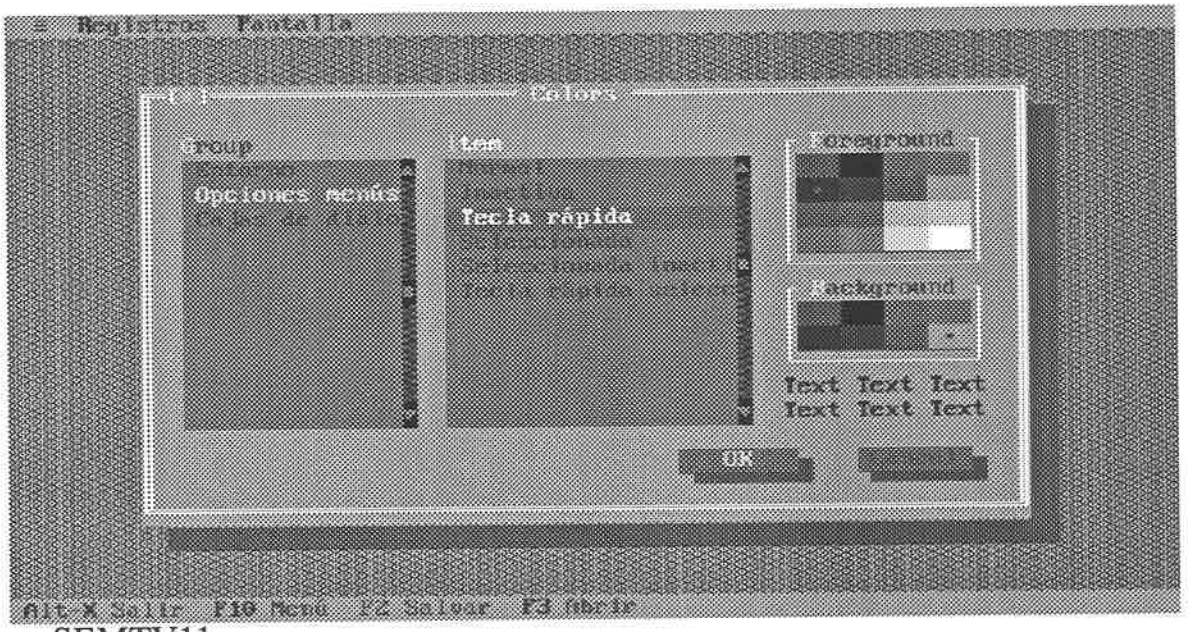


SEMTV11

En Turbo Vision las vistas al dibujarse en pantalla nunca utilizan directamente códigos de color, sino que utilizan paletas. Podemos imaginar una paleta como una matriz de enteros, donde cada uno de los elementos almacena un código. Imaginemos una caja de diálogo, en la que existen elementos tales como el borde, botones, etiquetas, etc. Una caja de diálogo es una vista, y como tal tiene su paleta de visualización. Cuando esta vista quiere dibujar su borde, por ejemplo, sabe que tiene que utilizar el elemento 1 de la paleta, es decir, de la matriz que comentaba anteriormente se toma el primer elemento. Sin embargo ese código obtenido de la paleta no es el color final con el que aparecerá visualizado el borde, ya que la caja de diálogo es en realidad una subvista, que está incluída en el despacho, que a su vez está incluído en la aplicación, que es la vista que agrupa a todas las demás. Por lo tanto el número obtenido del primer elemento de la paleta de la caja de diálogo nos servirá para obtener el código del elemento adecuado de la paleta del despacho, código que a su vez representa el elemento que habrá que tomar de la paleta de la aplicación, que finalmente representa el color con el que aparecerá la entidad. Este esquema de paletas de colores permite el que una misma vista se visualice en un color u otro dependiendo de la vista en que se halle incluída. Por ejemplo, una tecla de selección rápida puede tener distinto color en un menú que en una caja de diálogo, sin embargo la paleta de esta subvista, apunta al mismo elemento de la paleta de la vista a la que pertenece, y este código es el que difiere de una vista a otra, de un menú a una caja de diálogo.

Es fácil saber el código o número de elemento de paleta asignado a cada subvista de una vista a partir de los archivos de cabecera, principalmente los correspondientes a menús, cajas de diálogo, vistas y aplicación.

Una vez tenemos esto claro prosigamos con nuestra caja de diálogo de selección de colores. En esta caja aparecerán cuatro partes bien diferenciadas: una a la izquierda indicando el grupo o vista, otra en el centro especificando la subvista de ese grupo, un tercero en la parte superior derecha mostrando el color de primer plano que tiene y una última en la parte inferior derecha mostrando el color de fondo. Una vista, como puede ser un menú, tiene distintos colores para sus subvistas, opciones normales, opciones desactivadas, teclas de selección rápida, etc. El menú es el grupo, las subvistas son los elementos de ese grupo. Para crear un objeto **TColorDialog** tendremos que facilitar dos parámetros: una paleta a modificar y una lista enlazada de grupos con sus subvistas, especificando los elementos que se van a permitir modificar. La paleta a modificar puede ser una paleta vacía, que después podemos asignar a una vista, o una ya existente, que se puede obtener por medio del método **getPalette**(). La lista enlazada la crearemos basándonos en las clases **TGroup** y **TItem**; la primera tan sólo necesita el nombre del grupo o vista, mientras que la segunda tomará el nombre de la subvista y el elemento que le corresponde en la paleta a modificar.



SEMTV11

Para que pueda ver un ejemplo de las distintas cajas de diálogo estándar comentadas, vamos a añadirlas a nuestra aplicación. Para ello vamos a añadir a la barra de menús una nueva opción, el menú de sistema, en el que tendremos dos opciones, cambio de directorio y edición de archivo. También modificaremos la línea de estado, para incorporar los comandos de carga y grabación de un archivo. Hecho esto tan sólo queda preparar los métodos correspondientes, **mCambioDirectorio**() crea una caja de diálogo para cambio de directorio, que ejecuta y destruye a continuación; **mEditarArchivo**() crea una caja de selección de archivo, y si se selecciona alguno abre una ventana de edición ocupando la extensión disponible actualmente en el despacho, que se obtiene por medio del método **getExtent**(); **mConfigurarColores**(), por último, crea una lista enlazada de grupos y subvistas que a continuación usa para crear una caja de diálogo de selección de colores y ejecutarla, redibujando todas las vistas si se aceptó el cambio efectuado.

```
        //
        // SEMTV11.CPP                  Se añaden cajas de diálogo estandar
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory
#define Uses_TCheckBoxes
#define Uses_TRadioButtons
#define Uses_TSItem
#define Uses_TButton
#define Uses_TChDirDialog
#define Uses_TFileDialog
#define Uses_TEditWindow
#define Uses_TColorDialog
#define Uses_TColorGroup
#define Uses_TColorItem

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

#include <String.H> // Para usar la función strcpy()

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203,
        cmCambioDirectorio = 204,
        cmEditarArchivo = 205 ;

        // Se define una estructura que nos sirva
        // para almacenar la información de la caja de diálogo

struct {
        char Titulo[51] ;
        char Autor[51] ;
        char Editorial[51] ;
        ushort Opciones ;
        ushort Tema ;

} Libro ;

ushort NVen = 1 ; // Número de la ventana de edición

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() ;


                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;
                void mCambioDirectorio() ;
                void mEditarArchivo() ;

};

        // Implementación del constructor
```

```
AplicacionEstandar::AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )

{
                // Se inicializan los miembros de la estructura

        strcpy( Libro . Titulo, "" ) ;
        strcpy( Libro . Autor, "" ) ;
        strcpy( Libro . Editorial, "" ) ;

        Libro . Opciones = Libro . Tema = 0 ;
}


        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                                * new TStatusItem( 0, kbAltF3, cmClose ) +
                                * new TStatusItem( 0, kbCtrlF4, cmClose ) +
                                * new TStatusItem( "~F2~ Salvar", kbF2, cmSave ) +
                        * new TStatusItem( "~F3~ Abrir", kbF3, cmEditarArchivo )
                                ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla

        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                        * new TSubMenu( "~_~", kbAltSpace ) +
                                * new TMenuItem( "~D~irectorio", cmCambioDirectorio, 0, hcNoContext, "" ) +
                                * new TMenuItem( "~E~ditar archivo", cmEditarArchivo, 0, hcNoContext, "" ) +

                        * new TSubMenu( "~R~egistros", kbAltR ) +

                                * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                                * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                                newLine() +
                                * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

                        * new TSubMenu( "~P~antalla", kbAltP ) +

                                * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                                * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                                        ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;

                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :
```

```
                        mConfigurarColores() ;
                        break ;

                case cmCambioDirectorio :

                        mCambioDirectorio() ;
                        break ;

                case cmEditarArchivo :

                        mEditarArchivo() ;
                        break ;

                default :   // Si no es ninguno de los comandos reconocidos

                        return ; // Regresa sin procesarlo
            }

            clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
            // Se define una caja de diálogo

        TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                               "Registro de Libros" ) ;

        TInputLine * Linea ; // Un puntero a un objeto TInputLine
        TLabel * Etiqueta ; // Etiqueta de la línea

            // Se crea una línea con una entrada de 50 caracteres como máximo

        Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 51 ) ;

            // Se define una etiqueta enlazado al campo anterior

        Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

        RegistroLibro -> insert( Linea ) ; // Se insertan el campo y la
        RegistroLibro -> insert( Etiqueta ) ;        // etiqueta en la caja

            // Se repite el proceso para los otros dos campos

        Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 51 ) ;
        Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;

        RegistroLibro -> insert( Linea ) ;
        RegistroLibro -> insert( Etiqueta ) ;

        Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 51 ) ;
        Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

        RegistroLibro -> insert( Linea ) ;
        RegistroLibro -> insert( Etiqueta ) ;

            // Se crea una lista histórica de editoriales

        RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;

            // Se define una caja de selecciones múltiples con
            // tres posibles campos

        TCheckBoxes * Opciones = new TCheckBoxes( TRect( 40, 3, 57, 6 ),
                                    new TSItem( "~I~lustrado",
                                    new TSItem( "~D~isquete",
                                    new TSItem( "~C~olección",
                                    0 )))) ;

            // Se asigna una etiqueta a la caja de selección

        Etiqueta = new TLabel( TRect( 40, 2, 55, 3 ), "~O~pciones", Opciones ) ;

            // Se insertan ambos controles

        RegistroLibro -> insert( Opciones ) ;
        RegistroLibro -> insert( Etiqueta ) ;

            // Se crea una caja de selección exclusiva con
            // tres posibles opciones

        TRadioButtons * Tema = new TRadioButtons( TRect( 40, 8, 57, 11 ),
                                    new TSItem( "~N~ovela",
                                    new TSItem( "~P~oesía",
                                    new TSItem( "In~f~ormática",
                                    0 )))) ;

            // Se asigna una etiqueta a la caja de selección

        Etiqueta = new TLabel( TRect( 40, 7, 55, 8 ), "Te~m~a", Tema ) ;

            // Se insertan ambos controles

        RegistroLibro -> insert( Tema ) ;
        RegistroLibro -> insert( Etiqueta ) ;
```

```
        // Se insertan un botón Vale y un botón Cancelar

    RegistroLibro -> insert( new TButton( TRect( 5, 12, 15, 14 ), "~V~ale", cmOK, bfDefault )) ;
    RegistroLibro -> insert( new TButton( TRect( 20, 12, 34, 14 ), "~C~ancelar", cmCancel, bfNormal )) ;

    RegistroLibro -> setData( & Libro ) ; // Se fijan los valores de los campos

        // Se comprueba que los controles insertandos coincidan
        // en tamaño con los campos definidos en la estructura Libro

    if( sizeof( Libro ) != RegistroLibro -> dataSize() )

        messageBox( "El tamaño de la estructura no coincide", mfOKButton | mfWarning ) ;

        // Se ejecuta la caja de diálogo obteniendo el código
        // del botón pulsado

    ushort Boton = deskTop->execView( RegistroLibro ) ;

    if( Boton == cmOK )     // Si se aceptó el registro de este libro
    {
            // Se recoge la información

        RegistroLibro -> getData( & Libro ) ;

            // A continuación se puede realizar cualquier
            // proceso con la información, como su almacenamiento
            // en un archivo
    }

    destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}

    // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
    messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

    // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
    messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

    // Implementación del método mConfigurarColores
```

```
void AplicacionEstandar::mConfigurarColores()
{
        // Se crea el grupo de colores a modificar

    TColorGroup & Grupo = * new TColorGroup( "Entorno" ) +
                          * new TColorItem( "Fondo", 1 ) +
                          * new TColorGroup( "Opciones menús" ) +
                          * new TColorItem( "Normal", 2 ) +
                          * new TColorItem( "Inactiva", 3 ) +
                          * new TColorItem( "Tecla rápida", 4 ) +
                          * new TColorItem( "Seleccionada", 5 ) +
                          * new TColorItem( "Seleccionada inactiva", 6 ) +
                          * new TColorItem( "Tecla rápida seleccionada", 7 ) +
                          * new TColorGroup( "Cajas de diálogo" ) +
                          * new TColorItem( "Fondo", 33 ) ;

        // Se crea una caja de diálogo de selección de colores,
        // modificando la paleta de la aplicación, que se obtiene
        // con el método miembro getPalette()

    TColorDialog * Colores = new TColorDialog( & getPalette(), &Grupo ) ;

        // Se ejecuta la caja de diálogo

    ushort Boton = deskTop -> execView( Colores ) ;

    if( Boton != cmCancel ) // Si no se canceló el cambio
    {
            // Redibuja el despacho completo

        deskTop -> setState( sfVisible, False ) ;
        redraw() ;
        deskTop -> setState( sfVisible, True ) ;
    }

    destroy( Colores ) ; // Libera la memoria de la caja de diálogo
}

    // Implementación del método mCambioDirectorio

void AplicacionEstandar::mCambioDirectorio()
{
        // Se crea la caja de diálogo de cambio de directorio

    TChDirDialog * Caja = new TChDirDialog( cdNormal, cmCambioDirectorio ) ;

    deskTop -> execView( Caja ) ; // Se ejecuta
```

```
        destroy( Caja ) ; // Y se destruye
}

        // Implementación del método mEditarArchivo

void AplicacionEstandar::mEditarArchivo()
{
                // Se crea la caja de diálogo para selección de archivo

        TFileDialog * Caja = new TFileDialog( "*.*", "Selección Archivo", "Archivo",
                        fdOKButton, cmEditarArchivo ) ;

        ushort Control = deskTop -> execView( Caja ) ; // Se ejecuta

        if( Control == cmFileOpen ) // Si se ha confirmado la selección
        {
                char Archivo[MAXPATH+1] ;

                Caja -> getFileName( Archivo ) ;

                TRect Extension = getExtent() ;

                Extension . b . y -= 2 ;

                TEditWindow * Editor = new TEditWindow( Extension, Archivo, NVen++ ) ;

                deskTop-> insert( Editor ) ;


        }

        destroy( Caja ) ; // Se libera el objeto
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```

STREAMS

Se denomina stream al flujo de datos que se establece entre un generador y un consumidor de información, siendo éstos un archivo en disco, una aplicación, o cualquier otro dispositivo. Los streams en Turbo Vision pueden tener múltiples aplicaciones, pero entre ellas destacan principalmente dos: ofrecer persistencia a los objetos Turbo Vision, principalmente las vistas, y facilitar el manejo de recursos.

Creación de objetos persistentes

Un objeto Turbo Vision, por lo que hemos visto hasta ahora, se crea, se utiliza y se destruye. A veces este proceso se repite durante una misma ejecución, por ejemplo creando la misma caja de diálogo varias veces para después de usarla destruirla de nuevo.

En el momento en que un objeto es capaz de almacenarse en un stream, y más concretamente en un stream en disco, se dice que el objeto es persistente, ya que su existencia va más allá del tiempo que dura la ejecución de una aplicación. Los objetos persistentes son los que permiten, por ejemplo, que al salir del IDE de Borland C++ y entrar de nuevo encontremos las mismas ventanas abiertas, con los mismos tamaños y colores, los mismos contenidos en las listas históricas, etc.

Para que un objeto sea persistente tiene que ser capaz, como he dicho antes, de almacenarse en un stream, y lógicamente de leerse de él, para lo que tiene que incorporar los métodos y operadores apropiados. Todas las clases Turbo Vision que tienen esta capacidad están derivadas de una u otra forma de **TStreamable**, una clase abstracta que define tres métodos virtuales puros, lo que quiere decir que cualquier clase que derive de ella tendrá que definirlos. **TStreamable** es base de todas las clase Turbo Vision que representan vistas, desde **TView** hasta nuestra propia aplicación. Todas estas clases definen, por lo tanto, los métodos **read**() y **write**() necesarios para su almacenamiento y recuperación.

Con el fin de facilitar el trabajo con streams Turbo Vision define una jerarquía de clases que parte de **pstream**, definiendo como derivadas clases de entrada, salida y entrada/salida. Para trabajar con archivos en disco usaremos la familia **fpstream**, similar a la jerarquía **fstream** definida en iostream.

Por lo tanto es posible escribir en un **pstream** casi cualquier objeto Turbo Vision, además de cualquier otra información, como puede ser un entero, una cadena, etc. Antes de poder realizar una operación en un stream con un objeto de una determinada clase, es necesario que comunicar a Turbo Vision que esa clase es susceptible de ser almacenada/recuperada en un stream, registrándola por medio de la macro **__link**, que necesita como parámetro el nombre de la clase, sustituyendo la **T** inicial por una **R**. Una vez hecho esto no tendremos más que abrir el stream, y escribir o leer de él.

Durante la ejecución de una aplicación de usuario final no es normal escribir en streams objetos tales como cajas de diálogo o menús, ya que este tipo de vistas estarán ya diseñadas previamente, bien en el mismo programa o en un archivo de recursos. ¿Qué objetos tiene sentido almacenar en este tipo de aplicaciones?, aquellos que cambian de una ejecución a otra. Si en esa aplicación el usuario puede alterar los colores de visualización, el tamaño y posición de ventanas, por ejemplo, esos son los datos a conservar, para que al entrar la próxima vez lo encuentré todo tal y como lo dejó, y no con los valores iniciales de la aplicación. El candidato perfecto, por lo tanto, es el despacho, el objeto de la clase **TDeskTop** que contiene nuestra aplicación. Este objeto es una vista, pero al tiempo es un grupo que contiene subvistas. ¿Qué ocurre entonces al almacenar esta entidad en un stream?, que todas las subvistas que en ese momento están en el despacho también se almacenan, conservando su posición y demás parámetros. Lo único que el despacho no conserva es la paleta de colores de la aplicación, ya que pertenece a la aplicación, y no al despacho. De cualquier forma es fácil obtener esta paleta por medio del método **getPalette**(), y acceder a su contenido, una cadena de bytes, para almacenarlo o restaurarlo.

En el listado SEMTV12.CPP puede ver como el programa comprueba, al entrar, si existe un archivo de configuración, y en caso afirmativo lee de él el despacho y la paleta de colores, en caso contrario procede a la inicialización normal de la aplicación. Al finalizar la ejecución se almacena en el mismo archivo el despacho y la paleta de colores, que han podido ser modificados. Fíjese en que se ha añadido a la clase el método **initDeskTop**(), para poder asi controlar la creación del despacho, que se realizará llamando al mismo método de la clase base o leyéndolo del archivo de configuración. Para leer y escribir la paleta de colores se ha accedido al miembro **data** que tiene cualquier objeto de la clase **TPalette**. Este miembro es una cadena de caracteres, el primero de los cuales indica la longitud de la paleta, es decir, el número de caracteres que contiene la cadena, sin contarse a sí mismo.

```
        //
        // SEMTV12.CPP                  Se graba y recupera el despacho
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory
#define Uses_TCheckBoxes
#define Uses_TRadioButtons
#define Uses_TSItem
#define Uses_TButton
#define Uses_TChDirDialog
#define Uses_TFileDialog
#define Uses_TEditWindow
#define Uses_TColorDialog
#define Uses_TColorGroup
#define Uses_TColorItem
#define Uses_ofpstream          // Escritura y lectura de objetos Turbo Vision en streams
#define Uses_ifpstream

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

#include <String.H> // Para usar la función strcpy()

__link (RStatusLine);           // Se registran todos los posibles objetos
__link (RMenuBar);              // a almacenar en el stream
__link (RDialog);
__link (RDeskTop);
__link (RLabel);
__link (RInputLine);
__link (RHistory);
__link (RCheckBoxes);
__link (RRadioButtons);
__link (RButton);
__link (RChDirDialog);
__link (RFileDialog);
__link (REditWindow);
__link (RColorDialog);
__link (RView);
__link (RWindow);
__link (RBackground);

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203,
        cmCambioDirectorio = 204,
        cmEditarArchivo = 205 ;

        // Se define una estructura que nos sirva
        // para almacenar la información de la caja de diálogo

struct {
        char Titulo[51] ;
        char Autor[51] ;
        char Editorial[51] ;
        ushort Opciones ;
        ushort Tema ;

} Libro ;

        // Archivo para leer la configuración del programa

static  ifpstream  LeeConfiguracion( "SEMTV12.CFG" ) ;

ushort NVen = 1 ; // Número de la ventana de edición

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
 {
        public :

                // Se define el constructor de la clase

            AplicacionEstandar() ;


                // Método para inicializar la línea de estado
```

```
        static TStatusLine * initStatusLine( TRect ) ;

                // Método para inicializar la barra de menú

        static TMenuBar * initMenuBar( TRect ) ;

                // Se añade un método para inicializar el despacho

        static TDeskTop * initDeskTop( TRect ) ;

                // Método para la gestión de eventos propios

        virtual void handleEvent( TEvent & ) ;

                // Estos son los métodos miembros a los que
                // se llamará dependiendo del comando recibido

        void mRegistroLibro() ;
        void mRegistroArticulo() ;
        void mConfigurarModo() ;
        void mConfigurarColores() ;
        void mCambioDirectorio() ;
        void mEditarArchivo() ;

} ;

        // Implementación del constructor

AplicacionEstandar::AplicacionEstandar() :

                TProgInit( & initStatusLine,
                        & initMenuBar,
                        & initDeskTop )

{
                // Se inicializan los miembros de la estructura

        strcpy( Libro . Titulo, "" ) ;
        strcpy( Libro . Autor, "" ) ;
        strcpy( Libro . Editorial, "" ) ;

        Libro . Opciones = Libro . Tema = 0 ;
}


        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                                * new TStatusItem( 0, kbAltF3, cmClose ) +
                                * new TStatusItem( 0, kbCtrlF4, cmClose ) +
                                * new TStatusItem( "~F2~ Salvar", kbF2, cmSave ) +
                                * new TStatusItem( "~F3~ Abrir", kbF3, cmEditarArchivo )
                              ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
                // Se define como área de menús la primera línea de pantalla

        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                        * new TSubMenu( "~_~", kbAltSpace ) +

                            * new TMenuItem( "~D~irectorio", cmCambioDirectorio, 0, hcNoContext, "" ) +
                            * new TMenuItem( "~E~ditar archivo", cmEditarArchivo, 0, hcNoContext, "" ) +

                        * new TSubMenu( "~R~egistros", kbAltR ) +

                            * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
                            * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
                            newLine() +
                            * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

                        * new TSubMenu( "~P~antalla", kbAltP ) +

                            * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
                            * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                              ) ;
}

        // Implementación del método initDeskTop

TDeskTop * AplicacionEstandar::initDeskTop( TRect Area )
```

```
{
        static TDeskTop Despacho( Area ) ; // Se crea un despacho vacío

        if( ! LeeConfiguracion ) // Si no existe el archivo de configuración

                        // se devuelve el despacho por defecto

                return TApplication::initDeskTop( Area ) ;

        else // Si el archivo de configuración existe
        {
                LeeConfiguracion >> Despacho ; // se lee el despacho de él

                return & Despacho ; // y se devuelve
        }
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;

                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :

                                mConfigurarModo() ;
                                break ;

                        case cmConfigurarColores :

                                mConfigurarColores() ;
                                break ;

                        case cmCambioDirectorio :

                                mCambioDirectorio() ;
                                break ;

                        case cmEditarArchivo :

                                mEditarArchivo() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;    // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
                // Se define una caja de diálogo

        TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                                "Registro de Libros" ) ;

        TInputLine * Linea ; // Un puntero a un objeto TInputLine
        TLabel * Etiqueta ; // Etiqueta de la línea

                // Se crea una línea con una entrada de 50 caracteres como máximo

        Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 51 ) ;

                // Se define una etiqueta enlazado al campo anterior

        Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

        RegistroLibro -> insert( Linea ) ; // Se insertan el campo y la
        RegistroLibro -> insert( Etiqueta ) ;       // etiqueta en la caja

                // Se repite el proceso para los otros dos campos

        Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 51 ) ;
        Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;

        RegistroLibro -> insert( Linea ) ;
```

```
RegistroLibro -> insert( Etiqueta ) ;

Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 51 ) ;
Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una lista histórica de editoriales

RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;

        // Se define una caja de selecciones múltiples con
        // tres posibles campos

TCheckBoxes * Opciones = new TCheckBoxes( TRect( 40, 3, 57, 6 ),
                                new TSItem( "~I~lustrado",
                                new TSItem( "~D~isquete",
                                new TSItem( "~C~olección",
                                0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 2, 55, 3 ), "~O~pciones", Opciones ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Opciones ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una caja de selección exclusiva con
        // tres posibles opciones

TRadioButtons * Tema = new TRadioButtons( TRect( 40, 8, 57, 11 ),
                                new TSItem( "~N~ovela",
                                new TSItem( "~P~oesía",
                                new TSItem( "In~f~ormática",
                                0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 7, 55, 8 ), "Te~m~a", Tema ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Tema ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se insertan un botón Vale y un botón Cancelar

        RegistroLibro -> insert( new TButton( TRect( 5, 12, 15, 14 ), "~V~ale", cmOK, bfDefault )) ;
        RegistroLibro -> insert( new TButton( TRect( 20, 12, 34, 14 ), "~C~ancelar", cmCancel, bfNormal )) ;

        RegistroLibro -> setData( & Libro ) ; // Se fijan los valores de los campos

                // Se comprueba que los controles insertandos coincidan
                // en tamaño con los campos definidos en la estructura Libro

        if( sizeof( Libro ) != RegistroLibro -> dataSize() )

                messageBox( "El tamaño de la estructura no coincide", mfOKButton | mfWarning ) ;

                // Se ejecuta la caja de diálogo obteniendo el código
                // del botón pulsado

        ushort Boton = deskTop->execView( RegistroLibro ) ;

        if( Boton == cmOK )     // Si se aceptó el registro de este libro
        {
                        // Se recoge la información

                RegistroLibro -> getData( & Libro ) ;

                        // A continuación se puede realizar cualquier
                        // proceso con la información, como su almacenamiento
                        // en un archivo
        }

        destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
```

```
{
        // Se crea el grupo de colores a modificar

    TColorGroup & Grupo = * new TColorGroup( "Entorno" ) +
                          * new TColorItem( "Fondo", 1 ) +
                         * new TColorGroup( "Opciones menús" ) +
                          * new TColorItem( "Normal", 2 ) +
                          * new TColorItem( "Inactiva", 3 ) +
                          * new TColorItem( "Tecla rápida", 4 ) +
                          * new TColorItem( "Seleccionada", 5 ) +
                          * new TColorItem( "Seleccionada inactiva", 6 ) +
                          * new TColorItem( "Tecla rápida seleccionada", 7 ) +
                         * new TColorGroup( "Cajas de diálogo" ) +
                          * new TColorItem( "Fondo", 33 ) ;

        // Se crea una caja de diálogo de selección de colores,
        // modificando la paleta de la aplicación, que se obtiene
        // con el método miembro getPalette()

    TColorDialog * Colores = new TColorDialog( & getPalette(), &Grupo ) ;

        // Se ejecuta la caja de diálogo

    ushort Boton = deskTop -> execView( Colores ) ;

    if( Boton != cmCancel ) // Si no se canceló el cambio
    {
            // Redibuja el despacho completo

        deskTop -> setState( sfVisible, False ) ;
        redraw() ;
        deskTop -> setState( sfVisible, True ) ;
    }

    destroy( Colores ) ; // Libera la memoria de la caja de diálogo
}

    // Implementación del método mCambioDirectorio

void AplicacionEstandar::mCambioDirectorio()
{
        // Se crea la caja de diálogo de cambio de directorio

    TChDirDialog * Caja = new TChDirDialog( cdNormal, cmCambioDirectorio ) ;

    deskTop -> execView( Caja ) ; // Se ejecuta

    destroy( Caja ) ; // Y se destruye
}

    // Implementación del método mEditarArchivo

void AplicacionEstandar::mEditarArchivo()
{
        // Se crea la caja de diálogo para selección de archivo

    TFileDialog * Caja = new TFileDialog( "*.*", "Selección Archivo", "Archivo", fdOKButton, cmEditarArchivo ) ;

    ushort Control = deskTop -> execView( Caja ) ; // Se ejecuta

    if( Control == cmFileOpen ) // Si se ha confirmado la selección
    {
        char Archivo[MAXPATH+1] ;

        Caja -> getFileName( Archivo ) ;

        TRect Extension = getExtent() ;
        Extension . b . y -= 2 ;

        TEditWindow * Editor = new TEditWindow( Extension, Archivo, NVen++ ) ;

        deskTop-> insert( Editor ) ;


    }

    destroy( Caja ) ; // Se libera el objeto
}


int main()
{
    AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        // Se optiene una referencia a la paleta de colores de la aplicación

    TPalette & PaletaColores = MiAplicacion . getPalette() ;

    if( LeeConfiguracion ) // Si se ha abierto el archivo de configuración
    {
            // Leemos la paleta de colores
        LeeConfiguracion . readBytes( PaletaColores . data, (*PaletaColores . data)+1 ) ;

        LeeConfiguracion . close() ; // Cerramos el archivo de configuración

            // Y redibujamos el despacho para actualizar los colores
```

```
        MiAplicacion . deskTop -> setState( sfVisible, False ) ;
        MiAplicacion . redraw() ;
        MiAplicacion . deskTop -> setState( sfVisible, True ) ;
}

MiAplicacion . run() ; // Y se ejecuta

        // Una vez salimos de la aplicación se crea el
        // archivo de configuración

ofpstream GrabaConfiguracion( "SEMTV12.CFG" ) ;

        // Se graba el despacho

GrabaConfiguracion << *(MiAplicacion . deskTop) ;

        // Se graba la paleta de colores

GrabaConfiguracion . writeBytes( PaletaColores . data, (*PaletaColores . data)+1 ) ;

GrabaConfiguracion . close() ; // Se cierra el archivo de configuración

return 0 ; // Fin del programa
}
```

Recursos Turbo Vision

Se denomina recurso a toda aquella información que es parte de una aplicación sin formar parte del código ejecutable. Son recursos, por lo tanto, las cadenas de texto que se utilizan para mostrar mensajes, la barra de menú, las cajas de diálogo, etc. Todos estos objetos están generados a partir de un código, pero ellos en sí mismos no son código.

Hasta ahora los recursos de nuestra aplicación ejemplo están mezclados con el código de la aplicación, lo que hace que su mantenimiento sea complejo, ya que para modificar una caja de diálogo, o traducir los textos a otra lengua, por ejemplo, es necesario editar y modificar el programa, para a continuación volver a ejecutarlo. Visto desde este punto, sería mucho más lógico que los recursos estuviesen separados del código, para así poder modificarlos sin necesidad de tocar el fuente, evitando así la posibilidad de introducir errores en una aplicación depurada y flexibilizando al máximo su adptación. Tal y como hemos visto antes la mayoría de los objetos Turbo Vision son susceptibles de ser almacenados en un stream, y ser leídos posteriormente por la misma u otra aplicación. Sin embargo la idea de almacenar todos nuestros recursos utilizando este sistema, que en principio puede parecer buena, quiza no sea la más adecuada a la hora de trabajar con una cantidad importante de objetos, ya que el acceso a estos streams es secuencial, mientras que un determinado recurso puede ser necesario en un momento dado de forma aleatoria. Por ello Turbo Vision incorpora, basándose en el uso de streams visto antes, las clases necesarias para almacenar y recuperar recursos de un stream de forma directa, accediendo a ellos por medio de una clave.

TResourceFile es la clase a partir de la cual podremos abrir, bien sea para escribir o leer, un fichero de recursos. Para crear un objeto de esta clase tendremos que pasar como parámetro un **fpstream**, abierto para lectura o escritura, según el caso. Hecho esto **TResourceFile** definirá una lista de recursos o la leerá del archivo, almacenándola en un objeto **TResourceCollection**, que es una lista de cadenas ordenadas alfabéticamente, para así facilitar su búsqueda rápida. Teniendo abierto el fichero podremos añadir un recurso utilizando el método **put**() o leerlo con el método **get**(), facilitando, en el primer caso, el objeto a almacenar y su clave, y en el segundo caso sólo su clave. Al leer un recurso de un **TResourceFile** hay que tener en cuenta que es necesario utilizar moldeadores para convertir el objeto al tipo deseado, ya que el método **get**() devuelve un puntero **void**.

En el listado SEMTV13.CPP puede ver como se definen dos objetos, dos barras de menú distintas, y se almacenan en un archivo de recursos llamado SEMTV.RES. Nuestra aplicación ejemplo, modificada, carga uno u otro menú dependiendo de que al ejecutarla se pase como parámetro o no la cadena **estandar**.

```
        //
        // SEMTV13.CPP                  Crea un fichero de recursos con dos menús
        //

#define Uses_TRect      // Clases necesarias para crear menús
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_TKeys

#define Uses_fpstream  // Stream de entrada/salida en disco

#define Uses_TResourceFile     // Para crear el fichero de recursos
#define Uses_TResourceCollection // Para mantener el índice de recursos

#include <Tv.H> // Se incluyen los archivos de cabecera adecuados

__link(RMenuBar);                // En el archivo se van a almacenar los menús
__link(RResourceCollection);     // y el índice de recursos

const   cmRegistroLibro = 200,           // Comandos a los que dará acceso el menú
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203,
        cmCambioDirectorio = 204,
        cmEditarArchivo = 205 ;

int main()
 {
TRect Area( 0, 0, 80, 1 ) ;          // Area de visualización del menú

                // Se define un menú estandar con todas las
                // opciones de la aplicación

TMenuBar * MenuEstandar =

    new TMenuBar( Area,
      * new TSubMenu( "~_~", kbAltSpace ) +

        * new TMenuItem( "~D~irectorio", cmCambioDirectorio, 0, hcNoContext, "" ) +
        * new TMenuItem( "~E~ditar archivo", cmEditarArchivo, 0, hcNoContext, "" ) +

   * new TSubMenu( "~R~egistros", kbAltR ) +

        * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
        * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
        newLine() +
        * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

   * new TSubMenu( "~P~antalla", kbAltP ) +

        * new TMenuItem( "~M~odo", cmConfigurarModo, kbAltM, hcNoContext, "Alt-M" ) +
        * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                ) ;

                // Se define un menú restringido con
                // sólo algunas de las opciones
```

```
        TMenuBar * MenuRestringido =

            new TMenuBar( Area,
          * new TSubMenu( "~R~egistros", kbAltR ) +

              * new TMenuItem( "~L~ibro", cmRegistroLibro, kbAltL, hcNoContext, "Alt-L" ) +
              * new TMenuItem( "~A~rtículo", cmRegistroArticulo, kbAltA, hcNoContext, "Alt-A" ) +
              newLine() +
              * new TMenuItem( "~S~alir", cmQuit, kbAltX, hcNoContext, "Alt-X" ) +

          * new TSubMenu( "~P~antalla", kbAltP ) +

              * new TMenuItem( "~C~olores", cmConfigurarColores, kbAltC, hcNoContext, "Alt-C" )
                    ) ;

            // Se crea el archivo donde se almacenarán los recursos

        fpstream * Archivo = new fpstream( "SEMTV.RES", ios::out|ios::binary ) ;

                // Se crea el índice de recursos

        TResourceFile * ArchivoRecursos = new TResourceFile( Archivo ) ;

                // Se insertan los dos menús en el archivo

        ArchivoRecursos -> put( MenuEstandar, "MenuEstandar" ) ;
        ArchivoRecursos -> put( MenuRestringido, "MenuRestringido" ) ;

        TObject::destroy( ArchivoRecursos ) ;    // Se libera la memoria
        TObject::destroy( MenuEstandar ) ;
        TObject::destroy( MenuRestringido ) ;

        Archivo -> close() ; // Se cierra el archivo

        return 0; // Fin del programa
}
```

```
        //
        // SEMTV14.CPP                  Se lee el menú de un archivo de recursos
        //

#define Uses_TApplication       // Se definen las clases que se van a usar
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TKeys
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_MsgBox  // Para usar la función global messageBox()
#define Uses_TDialog
#define Uses_TDeskTop
#define Uses_TLabel
#define Uses_TInputLine
#define Uses_THistory
#define Uses_TCheckBoxes
#define Uses_TRadioButtons
#define Uses_TSItem
#define Uses_TButton
#define Uses_TChDirDialog
#define Uses_TFileDialog
#define Uses_TEditWindow
#define Uses_TColorDialog
#define Uses_TColorGroup
#define Uses_TColorItem
#define Uses_ofpstream          // Escritura y lectura de objetos Turbo Vision en streams
#define Uses_ifpstream
#define Uses_fpstream
#define Uses_TResourceFile
#define Uses_TResourceCollection

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <StdLib.H>  // Para usar la función random()

#include <String.H> // Para usar la función strcpy()

__link (RStatusLine);            // Se registran todos los posibles objetos
__link (RMenuBar);               // a almacenar en el stream
__link (RDialog);
__link (RDeskTop);
__link (RLabel);
__link (RInputLine);
__link (RHistory);
__link (RCheckBoxes);
__link (RRadioButtons);
__link (RButton);
__link (RChDirDialog);
__link (RFileDialog);
```

```
__link (REditWindow);
__link (RColorDialog);
__link (RView);
__link (RWindow);
__link (RBackground);
__link (RResourceCollection);

        // Se definen las constantes que representan nuestro nuevos comandos

const   cmRegistroLibro = 200,
        cmRegistroArticulo = 201,
        cmConfigurarModo = 202,
        cmConfigurarColores = 203,
        cmCambioDirectorio = 204,
        cmEditarArchivo = 205 ;

        // Se define una estructura que nos sirva
        // para almacenar la información de la caja de diálogo

struct {
        char Titulo[51] ;
        char Autor[51] ;
        char Editorial[51] ;
        ushort Opciones ;
        ushort Tema ;

} Libro ;

        // Archivo para leer la configuración del programa

static  ifpstream  LeeConfiguracion( "SEMTV12.CFG" ) ;

ushort NVen = 1 ; // Número de la ventana de edición

char Menu ;     // Tomara 0 ó 1 según el tipo de menú a usar

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
 {
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() ;

                        // Método para inicializar la línea de estado

                static TStatusLine * initStatusLine( TRect ) ;

                        // Método para inicializar la barra de menú

                static TMenuBar * initMenuBar( TRect ) ;

                        // Se añade un método para inicializar el despacho

                static TDeskTop * initDeskTop( TRect ) ;

                        // Método para la gestión de eventos propios

                virtual void handleEvent( TEvent & ) ;

                        // Estos son los métodos miembros a los que
                        // se llamará dependiendo del comando recibido

                void mRegistroLibro() ;
                void mRegistroArticulo() ;
                void mConfigurarModo() ;
                void mConfigurarColores() ;
                void mCambioDirectorio() ;
                void mEditarArchivo() ;

} ;

        // Implementación del constructor

AplicacionEstandar::AplicacionEstandar() :

                        TProgInit( & initStatusLine,
                                & initMenuBar,
                                & initDeskTop )

{
                // Se inicializan los miembros de la estructura

        strcpy( Libro . Titulo, "" ) ;
        strcpy( Libro . Autor, "" ) ;
        strcpy( Libro . Editorial, "" ) ;

        Libro . Opciones = Libro . Tema = 0 ;
}
```

```
        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

                // Se crea una línea de estado y se devuelve

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F10~ Menú", kbF10, cmMenu ) +
                                * new TStatusItem( 0, kbAltF3, cmClose ) +
                                * new TStatusItem( 0, kbCtrlF4, cmClose ) +
                                * new TStatusItem( "~F2~ Salvar", kbF2, cmSave ) +
                                * new TStatusItem( "~F3~ Abrir", kbF3, cmEditarArchivo )
                              ) ;
}

        // Implementación del método initStatusBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect )
{
                // Se abre el archivo de recursos

        fpstream * Archivo = new fpstream( "SEMTV.RES", ios::in | ios::binary ) ;

        if( ! Archivo ) // Si no se puede abrir
        {
                messageBox( "No se puede abrir SEMTV.RES", mfOKButton | mfWarning )
;
                exit(-1);
        }

                // Obtiene el índice de recursos

        TResourceFile * ArchivoRecursos = new TResourceFile( Archivo ) ;

        if( Menu ) // Según el valor de Menu devuelve

                        // un menú

                return (TMenuBar *) ArchivoRecursos -> get( "MenuEstandar" ) ;

        else
                        // u otro

                return (TMenuBar *) ArchivoRecursos -> get( "MenuRestringido" ) ;

}

        // Implementación del método initDeskTop

TDeskTop * AplicacionEstandar::initDeskTop( TRect Area )
{
        static TDeskTop Despacho( Area ) ; // Se crea un despacho vacío

        if( ! LeeConfiguracion ) // Si no existe el archivo de configuración

                        // se devuelve el despacho por defecto

                return TApplication::initDeskTop( Area ) ;

        else // Si el archivo de configuración existe
        {
                LeeConfiguracion >> Despacho ; // se lee el despacho de él

                return & Despacho ; // y se devuelve
        }
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmRegistroLibro :

                                mRegistroLibro() ;      // Llama al método adecuado
                                break ;

                        case cmRegistroArticulo :

                                mRegistroArticulo() ;
                                break ;

                        case cmConfigurarModo :
```

```
                        mConfigurarModo() ;
                        break ;

                case cmConfigurarColores :

                        mConfigurarColores() ;
                        break ;

                case cmCambioDirectorio :

                        mCambioDirectorio() ;
                        break ;

                case cmEditarArchivo :

                        mEditarArchivo() ;
                        break ;

                default :   // Si no es ninguno de los comandos reconocidos

                        return ; // Regresa sin procesarlo
            }

            clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}

        // Implementación del método mRegistroLibro

void AplicacionEstandar::mRegistroLibro()
{
            // Se define una caja de diálogo

        TDialog * RegistroLibro = new TDialog( TRect( 10, 5, 70, 20 ),
                                               "Registro de Libros" ) ;

        TInputLine * Linea ; // Un puntero a un objeto TInputLine
        TLabel * Etiqueta ; // Etiqueta de la línea

            // Se crea una línea con una entrada de 50 caracteres como máximo

        Linea = new TInputLine( TRect( 2, 3, 35, 4 ), 51 ) ;

            // Se define una etiqueta enlazado al campo anterior

        Etiqueta = new TLabel( TRect( 2, 2, 35, 3 ), "~T~ítulo", Linea ) ;

        RegistroLibro -> insert( Linea ) ; // Se insertan el campo y la

RegistroLibro -> insert( Etiqueta ) ;       // etiqueta en la caja

        // Se repite el proceso para los otros dos campos

Linea = new TInputLine( TRect( 2, 6, 35, 7 ), 51 ) ;
Etiqueta = new TLabel( TRect( 2, 5, 35, 6 ), "~A~utor", Linea ) ;

RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

Linea = new TInputLine( TRect( 2, 9, 35, 10 ), 51 ) ;
Etiqueta = new TLabel( TRect( 2, 8, 35, 9 ), "~E~ditorial", Linea ) ;

RegistroLibro -> insert( Linea ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una lista histórica de editoriales

RegistroLibro -> insert( new THistory( TRect( 35, 9, 38, 10 ), Linea, 0 ) ) ;

        // Se define una caja de selecciones múltiples con
        // tres posibles campos

TCheckBoxes * Opciones = new TCheckBoxes( TRect( 40, 3, 57, 6 ),
                                  new TSItem( "~I~lustrado",
                                  new TSItem( "~D~isquete",
                                  new TSItem( "~C~olección",
                                  0 )))) ;

        // Se asigna una etiqueta a la caja de selección

Etiqueta = new TLabel( TRect( 40, 2, 55, 3 ), "~O~pciones", Opciones ) ;

        // Se insertan ambos controles

RegistroLibro -> insert( Opciones ) ;
RegistroLibro -> insert( Etiqueta ) ;

        // Se crea una caja de selección exclusiva con
        // tres posibles opciones

TRadioButtons * Tema = new TRadioButtons( TRect( 40, 8, 57, 11 ),
                                  new TSItem( "~N~ovela",
                                  new TSItem( "~P~oesía",
                                  new TSItem( "In~f~ormática",
                                  0 )))) ;

        // Se asigna una etiqueta a la caja de selección
```

```
        Etiqueta = new TLabel( TRect( 40, 7, 55, 8 ), "Te~m~a", Tema ) ;

                // Se insertan ambos controles

        RegistroLibro -> insert( Tema ) ;
        RegistroLibro -> insert( Etiqueta ) ;

                // Se insertan un botón Vale y un botón Cancelar

        RegistroLibro -> insert( new TButton( TRect( 5, 12, 15, 14 ), "~V~ale", cmOK, bfDefault )) ;
        RegistroLibro -> insert( new TButton( TRect( 20, 12, 34, 14 ), "~C~ancelar", cmCancel, bfNormal )) ;

        RegistroLibro -> setData( & Libro ) ; // Se fijan los valores de los campos

                // Se comprueba que los controles insertandos coincidan
                // en tamaño con los campos definidos en la estructura Libro

        if( sizeof( Libro ) != RegistroLibro -> dataSize() )

                messageBox( "El tamaño de la estructura no coincide", mfOKButton | mfWarning ) ;

                // Se ejecuta la caja de diálogo obteniendo el código
                // del botón pulsado

        ushort Boton = deskTop->execView( RegistroLibro ) ;

        if( Boton == cmOK )     // Si se aceptó el registro de este libro
        {
                        // Se recoge la información

                RegistroLibro -> getData( & Libro ) ;

                        // A continuación se puede realizar cualquier
                        // proceso con la información, como su almacenamiento
                        // en un archivo
        }

        destroy( RegistroLibro ) ;          // Liberar toda la memoria asociada
}

        // Implementación del método mRegistroArticulo

void AplicacionEstandar::mRegistroArticulo()
{
        messageBox( "Opción de Registro de Artículos", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarModo

void AplicacionEstandar::mConfigurarModo()
{
        messageBox( "Opción de Configuración de Modo", mfOKButton | mfInformation ) ;
}

        // Implementación del método mConfigurarColores

void AplicacionEstandar::mConfigurarColores()
{
                // Se crea el grupo de colores a modificar

        TColorGroup & Grupo = * new TColorGroup( "Entorno" ) +
                                * new TColorItem( "Fondo", 1 ) +
                               * new TColorGroup( "Opciones menús" ) +
                                * new TColorItem( "Normal", 2 ) +
                                * new TColorItem( "Inactiva", 3 ) +
                                * new TColorItem( "Tecla rápida", 4 ) +
                                * new TColorItem( "Seleccionada", 5 ) +
                                * new TColorItem( "Seleccionada inactiva", 6 ) +
                                * new TColorItem( "Tecla rápida seleccionada", 7 ) +
                               * new TColorGroup( "Cajas de diálogo" ) +
                                * new TColorItem( "Fondo", 33 ) ;

                // Se crea una caja de diálogo de selección de colores,
                // modificando la paleta de la aplicación, que se obtiene
                // con el método miembro getPalette()

        TColorDialog * Colores = new TColorDialog( & getPalette(), &Grupo ) ;

                // Se ejecuta la caja de diálogo

        ushort Boton = deskTop -> execView( Colores ) ;

        if( Boton != cmCancel ) // Si no se canceló el cambio
        {
                        // Redibuja el despacho completo

                deskTop -> setState( sfVisible, False ) ;
                redraw() ;
                deskTop -> setState( sfVisible, True ) ;
        }

        destroy( Colores ) ; // Libera la memoria de la caja de diálogo
}

        // Implementación del método mCambioDirectorio

void AplicacionEstandar::mCambioDirectorio()
{
```

```
                // Se crea la caja de diálogo de cambio de directorio

        TChDirDialog * Caja = new TChDirDialog( cdNormal, cmCambioDirectorio ) ;

        deskTop -> execView( Caja ) ; // Se ejecuta

        destroy( Caja ) ; // Y se destruye
}

        // Implementación del método mEditarArchivo

void AplicacionEstandar::mEditarArchivo()
{
                // Se crea la caja de diálogo para selección de archivo

        TFileDialog * Caja = new TFileDialog( "*.*", "Selección Archivo", "Archivo", fdOKButton, cmEditarArchivo
) ;

        ushort Control = deskTop -> execView( Caja ) ; // Se ejecuta

        if( Control == cmFileOpen ) // Si se ha confirmado la selección
        {
                char Archivo[MAXPATH+1] ;

                Caja -> getFileName( Archivo ) ;

                TRect Extension = getExtent() ;
                Extension . b . y -= 2 ;

                TEditWindow * Editor = new TEditWindow( Extension, Archivo, NVen++ ) ;

                deskTop-> insert( Editor ) ;


        }

        destroy( Caja ) ; // Se libera el objeto
}


int main( int, char * argv[] )
{
        Menu = ! strcmp( strupr(argv[1]), "ESTANDAR" ) ;

        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

                // Se optiene una referencia a la paleta de colores de la aplicación

        TPalette & PaletaColores = MiAplicacion . getPalette() ;

        if( LeeConfiguracion ) // Si se ha abierto el archivo de configuración
        {
                        // Leemos la paleta de colores
                LeeConfiguracion . readBytes( PaletaColores . data, (*PaletaColores . data)+1 ) ;

                LeeConfiguracion . close() ; // Cerramos el archivo de configuración

                        // Y redibujamos el despacho para actualizar los colores

                MiAplicacion . deskTop -> setState( sfVisible, False ) ;
                MiAplicacion . redraw() ;
                MiAplicacion . deskTop -> setState( sfVisible, True ) ;
        }

        MiAplicacion . run() ; // Y se ejecuta

                // Una vez salimos de la aplicación se crea el
                // archivo de configuración

        ofpstream GrabaConfiguracion( "SEMTV12.CFG" ) ;

                // Se graba el despacho

        GrabaConfiguracion << *(MiAplicacion . deskTop) ;

                // Se graba la paleta de colores

        GrabaConfiguracion . writeBytes( PaletaColores . data, (*PaletaColores . data)+1 ) ;

        GrabaConfiguracion . close() ; // Se cierra el archivo de configuración

        return 0 ; // Fin del programa
}
```

Ventanas

Las cajas de diálogo que hemos utilizado en algunas de las opciones del menú de ejemplo anteriores, tanto las definidas por nosotros como las estandar, derivan de la clase **TWindow**, que a su vez deriva de **TView**. Un objeto **TWindow**, una ventana, es un grupo preparado para contener vistas en un recinto rectángular, que puede ser desplazado y redimensionado sin necesidad de escribir código alguno. Una caja de diálogo no es mas que una ventana especializada y con unas ciertas características.

Definir e insertar una ventana

A la hora de crear una ventana tenemos dos opciones, utilizar directamente la clase **TWindow** o derivar una nueva clase más especializada. El constructor de esta clase necesita como parámetros un objeto **TRect** indicando el área de la ventana, una cadena especificando el título y por último un número de ventana, que aparecerá en su marco.

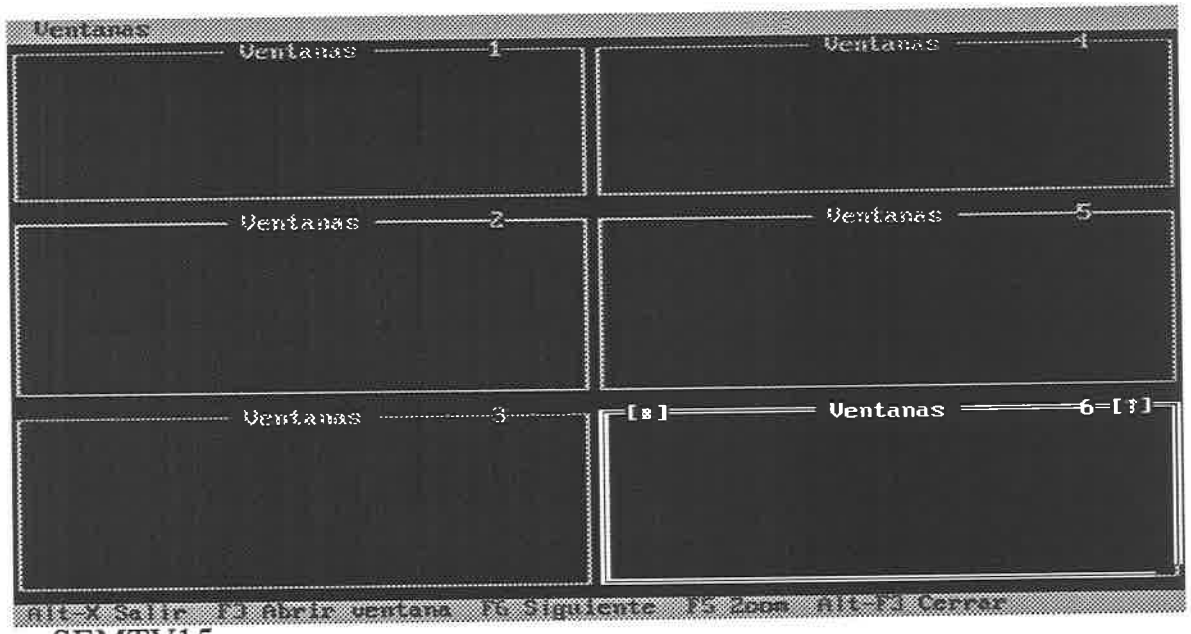
Una vez creada una ventana se puede insertar en el despacho con el método **insert** que ya utilizamos anteriormente. Hay que tener en cuenta que una ventana, por defecto, no tiene nada en el interior y no responde a evento alguno salvo el de los controles que tiene en su marco, minimizar, maximizar, cerrar, desplazar y redimensionar. Todos estos eventos afectan a la visualización de la ventana, pero no a la funcionalidad que nosotros deseemos darle.

Si el indicador **ofTileable** del campo **options** de la ventana está activado, el despacho puede redistribuir las ventanas que actualmente hay visualizadas, disponiéndolas en forma de cascada, con el método **cascade**(), o de mosaico, con el método **tile**(). Además, una ventana sabe responder automáticamente a los comandos **cmNext**, **cmZoom** y **cmClose**.

En el listado SEMTV15 tiene un ejemplo en el que se crean ventanas y se manipulan con los métodos y comandos mencionados.

```
        //
        // SEMTV15.CPP                  Creación y manipulación de ventanas
        //

#define Uses_TApplication
#define Uses_TWindow
#define Uses_TDeskTop
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_TRect
#define Uses_TKeys
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem

#include <Tv.H>         // Se incluye el archivo de cabecera Turbo Vision

const cmNueva = 201,
    cmCascada = 202,
    cmMosaico = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() ;

                static TStatusLine * initStatusLine( TRect ) ; // Línea de estado

                static TMenuBar * initMenuBar( TRect ) ; // Creación del menú

                        // Método para la gestión de eventos

                virtual void handleEvent( TEvent & ) ;

                        // Métodos de respuesta a eventos

                void mNueva() ;

                void mCascada() ;

                void mMosaico() ;

} ;

        // Se implementa el constructor de la clase

AplicacionEstandar::AplicacionEstandar() :
                        TProgInit( & initStatusLine,
                                   & initMenuBar,
                                   & initDeskTop )

{
}

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F3~ Abrir ventana", kbF3, cmNueva ) +
                                * new TStatusItem( "~F6~ Siguiente", kbF6, cmNext ) +
                                * new TStatusItem( "~F5~ Zoom", kbF5, cmZoom ) +
                        * new TStatusItem( "~Alt-F3~ Cerrar", kbAltF3, cmClose )
                                ) ;
}

        // Implementación del método initMenuBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                                * new TSubMenu( "~V~entanas", kbAltR ) +

                                        * new TMenuItem( "~N~ueva", cmNueva, 0, hcNoContext, "" ) +
                                        * new TMenuItem( "~C~ascada", cmCascada, 0, hcNoContext, "" ) +
                                        * new TMenuItem( "~M~osaico", cmMosaico, 0, hcNoContext, "" )
                                ) ;
}

        // Implementación del método handleEvent
```

```
void AplicacionEstandar::handleEvent( TEvent & Evento )
{
        // Primero se ha de procesar el evento por parte
        // del gestor de eventos de nuestra clase base

    TApplication::handleEvent( Evento ) ;

    if( Evento . what == evCommand )        // Si el evento es un comando
    {
        switch( Evento . message . command ) // Según el comando que sea
        {
            case cmNueva :

                mNueva() ;      // Llama al método adecuado
                break ;

            case cmCascada :

                mCascada() ;
                break ;

            case cmMosaico :

                mMosaico() ;
                break ;

            default :   // Si no es ninguno de los comandos reconocidos

                return ; // Regresa sin procesarlo
        }

        clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
    }
}


    // Implementación del método de respuesta al comando cmNueva

void AplicacionEstandar::mNueva()
{
    static ushort nVentana = 1 ;

        // Crear una ventana

    TWindow * Ventana = new TWindow( TRect( 5, 5, 35, 15 ), "Ventanas", nVentana++ ) ;

        // Permitir el poner la ventana en forma de cascada o mosaico

        Ventana -> options |= ofTileable ;

        deskTop -> insert( Ventana ) ; // Insertarla en el despacho
}

        // Implementación del método de respuesta al comando cmCascada

void AplicacionEstandar::mCascada()
{
        deskTop -> cascade( deskTop -> getExtent() ) ;
}

        // Implementación del método de respuesta al comando cmMosaico

void AplicacionEstandar::mMosaico()
{
        deskTop -> tile( deskTop -> getExtent() ) ;
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```



SEMTV15

Mantener el interior de una ventana

Como comenté anteriormente, no es posible visualizar nada directamente en una ventana, ya que este objeto es en realidad un grupo preparado para almacenar vistas, que sean capaces de dibujarse a sí mismas. Por lo tanto podríamos insertar objetos tales como botones, líneas de entrada de datos, etc. También podemos optar por crear nuestras propias vistas, derivando una nueva clase de **TView**, que contiene los métodos genéricos de todas las vistas.

Cualquier clase derivada de **TView** debe definir básicamente dos métodos: el constructor, donde se fijará el área o límites de la vista y se activarán o desactivarán ciertos indicadores, y el método virtual **draw**(), que será llamado cada vez que la vista tenga que ser dibujada.

Un objeto **TView**, o derivado, tiene entre sus miembros tres campos interesantes, ya que por medio de ellos se alteran ciertos comportamientos de la vista. El campo **dragMode** está compuesto por una serie de indicadores que establecen si la ventana puede

o no ser redimensionada, si puede ser movida de sitio en el despacho y si puede o no sobrepasar los límites visibles. El campo **growMode** se compone de unos indicadores que establecen el cambio que se ha de efectuar en la vista cada vez que el tamaño de la vista padre cambie; por medio de ellos es posible, por ejemplo, que al cambiar el tamaño de una ventana lo haga tambien el espacio de visualización que hay en su interior, o lo que es lo mismo, que las subvistas que contiene también cambien. Por último tenemos el campo **options,** por medio del cual se pueden establecer características tales como si la vista debe tener o no un marco alrededor, si es posible reposicionarlo con los comandos **cascade()** y **tile()** del despacho, si al insertarlos en otra vista deben aparecer centrados, etc. En general, cualquier clase derivada de **TView** deberá, en su constructor, dar el valor adecuado a estos indicadores.

El método **draw()** de una vista es el único lugar de una aplicación Turbo Vision donde se da salida a información por la pantalla. Todas las vistas deben tener definido este método, que debe seguir unas reglas de comportamiento para que la visualización se realice de forma correcta. Ante todo hay que tener en cuenta que una vista se tiene que hacer cargo, en todo momento, de la extensión que tiene adjudicada. Turbo Vision no realiza comprobaciones respecto a esto, por lo que si una vista escribe sólo en parte de su extensión, al dar salida a la pantalla el resto del área puede contener cualquier cosa. El método **draw()** tiene que sobreescribir en cada momento un área que está definida por el **size**, cuyo miembro **x** contiene el número de columnas y el miembro **y** el número de filas del área. En aquellas zonas donde no se vaya a representar información alguna habrá que rellenar con espacios. Para facilitar esto nos serviremos de un objeto de la clase **TDrawBuffer**, que es una representación en memoria de lo que debe aparecer en pantalla, donde podremos realizar cualquier manipulación previa a la visualización final. Para escribir en un objeto de este tipo disponemos de varios métodos, entre ellos **moveChar()**, que nos permite escribir un número determinado de veces un mismo carácter a partir de una posición especificada y en un color designado por nosotros; el método **moveStr()**, por su parte, nos facilita el insertar una cadena de caracteres en el buffer a partir de una posición indicada y con un color seleccionado por nosotros. Para escribir un buffer en su posición definitiva en la vista nos serviremos del método **writeLine()**, al que indicaremos el tamaño y posición del recuadro donde se va a volcar el buffer, pasados como cuatro parámetros enteros, y el buffer a escribir.

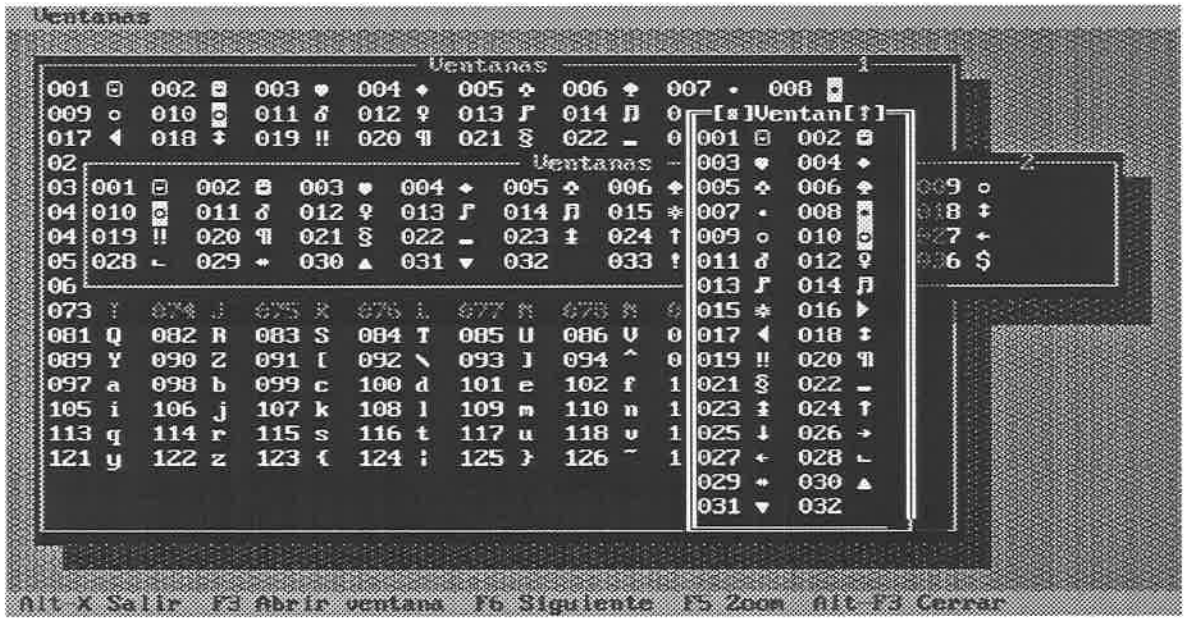
A la hora de escribir el contenido de un buffer hemos de tener en cuenta el hecho de que no utilizamos códigos de color, sino índices de la paleta de colores de la vista a la que pertenecemos. Por ello a la hora de usar un color utilizaremos el método **getColor()** para obtener el color final, pasando como parámetro el índice de la paleta de la vista padre.

En el listado SEMTV16.CPP puede ver como se deriva una nueva clase a partir de **TView**, a la que llamo **Contenido**, que implementa tan sólo el constructor y el método virtual **draw()**. El constructor se limita a pasar al constructor de **TView** el área de visualización de la vista y a activar los indicadores **gfGrowHiX** y **gfGrowHiY** del campo **growMode**, para que al redimensionar la ventana también cambie de tamaño nuestra vista, ya que queremos que en todo momento ocupe por completo la ventana. El método **draw()**, por su parte, visualiza en el área correspondiente a la vista una tabla de códigos ASCII y su representación decimal.

```
        //
        // SEMTV16.CPP              Se añade un contenido a las ventanas
        //

#define Uses_TApplication
#define Uses_TWindow
#define Uses_TDeskTop
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_TRect
#define Uses_TKeys
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TView

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <String.H> // Para usar funciones de cadena

#include <Stdio.H> // Para usar la funcion sprintf()

const cmNueva = 201,
    cmCascada = 202,
    cmMosaico = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase
                AplicacionEstandar() ;

                static TStatusLine * initStatusLine( TRect ) ; // Línea de estado

                static TMenuBar * initMenuBar( TRect ) ; // Creación del menú

                        // Método para la gestión de eventos

                virtual void handleEvent( TEvent & ) ;

                        // Métodos de respuesta a eventos

                void mNueva() ;

                void mCascada() ;

                void mMosaico() ;

} ;

        // Se crea un nuevo tipo de vista a la que se llama Contenido

class Contenido : public TView
{
        public :

                Contenido( TRect ) ; // Constructor

                void draw() ; // Método encargado de dibujar la vista
} ;

        // Se implementa el constructor de la clase

AplicacionEstandar::AplicacionEstandar() :
                        TProgInit( & initStatusLine,
                                & initMenuBar,
```

```
                                & initDeskTop )

{
}

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F3~ Abrir ventana", kbF3, cmNueva ) +
                                * new TStatusItem( "~F6~ Siguiente", kbF6, cmNext ) +
                                * new TStatusItem( "~F5~ Zoom", kbF5, cmZoom ) +
                        * new TStatusItem( "~Alt-F3~ Cerrar", kbAltF3, cmClose )
                                ) ;
}

        // Implementación del método initMenuBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                                * new TSubMenu( "~V~entanas", kbAltR ) +

                                        * new TMenuItem( "~N~ueva", cmNueva, 0, hcNoContext, "" ) +
                                        * new TMenuItem( "~C~ascada", cmCascada, 0, hcNoContext, "" ) +
                                        * new TMenuItem( "~M~osaico", cmMosaico, 0, hcNoContext, "" )
                                ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmNueva :

                                mNueva() ;      // Llama al método adecuado
                                break ;

                        case cmCascada :

                                mCascada() ;
                                break ;

                        case cmMosaico :

                                mMosaico() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}


        // Implementación del método de respuesta al comando cmNueva

void AplicacionEstandar::mNueva()
{
        static ushort nVentana = 1 ;

                // Crear una ventana

        TWindow * Ventana = new TWindow( TRect( 5, 5, 35, 15 ), "Ventanas", nVentana++ ) ;

                // Permitir el poner la ventana en forma de cascada o mosaico

        Ventana -> options |= ofTileable ;

                // Se obtiene la extensión actual de la ventana

        TRect Extension = Ventana -> getExtent() ;

                // Se reduce para que la vista que se va a insertar
                // ocupe todo excepto el marco delimitador
```

```
        Extension . grow( -1, -1 ) ;

                // Se crea un contenido y se inserta

        Ventana -> insert( new Contenido( Extension ) ) ;

        deskTop -> insert( Ventana ) ; // Insertarla en el despacho
}

        // Implementación del método de respuesta al comando cmCascada

void AplicacionEstandar::mCascada()
{
        deskTop -> cascade( deskTop -> getExtent() ) ;
}

        // Implementación del método de respuesta al comando cmMosaico

void AplicacionEstandar::mMosaico()
{
        deskTop -> tile( deskTop -> getExtent() ) ;
}

        // Implementación del constructor de la clase Contenido

Contenido::Contenido( TRect Limites ) : TView( Limites )
{
                // Se activan los indicadores gfGrowHiX y gfGrowHiY,
                // para que al redimensionar la ventana automáticamente
                // se redimensione el tamaño de Contenido

        growMode = gfGrowHiX | gfGrowHiY ;
}

        // Implementación del método draw

void Contenido::draw()
{
        ushort Color = getColor( 6 ) ; // Se obtiene el color de texto normal

        TDrawBuffer Buffer ; // Buffer para preparar la salida

        int Codigo = 1 ; // Contador de códigos ASCII

                // Recorrer todas las líneas que tiene la ventana

        for( int i = 0 ; i < size . y ; i++ )
        {
                int Columnas = size . x / 7 ; // Columnas de códigos posibles

                char Cadena[ 256 ] ; // Para formatear la salida

                strcpy( Cadena, "" ) ; // En principio la cadena está vacía

                        // Hasta llenar la ventana o llegar al código 127

                for( int j = 0 ; j < Columnas && Codigo < 128 ; j++ )
                {
                        char Buffer[ 10 ] ;

                                // Formatear la salida

                        sprintf( Buffer, "%03d %c  ", Codigo++, Codigo ) ;

                        strcat( Cadena, Buffer ) ; // Y añadirla a la Cadena
                }

                        // Limpiar el contenido del buffer

                Buffer . moveChar( 0, ' ', Color, size . x ) ;

                        // Añadirle la cadena a mostrar

                Buffer . moveStr( 0, Cadena, Color ) ;

                        // Escribir el buffer en la vista

                writeLine( 0, i, size . x, 1, Buffer ) ;
        }
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```



SEMTV16

Barras de desplazamiento

En ocasiones, el contenido que se desea representar en una ventana excede de las dimensiones de ésta, no siendo posible, por lo tanto, representar esa información de una sola vez. Una solución, la más utilizada en los entornos actuales, es dotar a esa ventana de barras de desplazamiento, por medio de las cuales sea posible desplazar la ventana por un área mucho mayor.

Para crear una vista dotada de barras de desplazamiento derivaremos una nueva clase a partir de **TScroller**, cuyo constructor necesita, además del área de la vista, dos punteros a objetos **TScrollBar**, definiendo el primero la barra de desplazamiento horizontal y el segundo la barra de desplazamiento vertical. Para crear un objeto **TScrollBar** lo más fácil es utilizar el método **standardScrollBar**(), al que tan sólo hemos de pasar como parámetro la opciones que definen la barra, **sbVertical** o **sbHorizontal**, según que la barra sea vertical u horizontal, y opcionalmente **sbHandleKeyboard**, para que la barra también responda a eventos de teclado.

El método **draw()** de una vista desplazable debe tener en cuenta que no debe dibujar el contenido desde su principio hasta donde pueda, según la extensión, sino desde la posición actual indicada por la barra de desplazamiento. Para ello contaremos, además del miembro **size** indicando el tamaño, con un miembro llamado **delta**, cuyo campo **x** nos facilita la primera columna que debe aparecer en la vista, mientras que el campo **y** hace lo propio con la línea. Este miembro, por lo tanto, nos indica el desplazamiento que debemos efectuar desde el principio del contenido a visualizar.

En el listado SEMTV17.CPP puede ver como a la ventana de códigos ASCII se ha añadido una barra de desplazamiento vertical, por medio de la cual es posible moverse por el contenido.

```
        //
        // SEMTV17.CPP                  Se añaden barras de desplazamiento
        //

#define Uses_TApplication
#define Uses_TWindow
#define Uses_TDeskTop
#define Uses_TMenuBar
#define Uses_TSubMenu
#define Uses_TMenuItem
#define Uses_TRect
#define Uses_TKeys
#define Uses_TStatusLine
#define Uses_TStatusDef
#define Uses_TStatusItem
#define Uses_TView
#define Uses_TScroller

#include <Tv.H>          // Se incluye el archivo de cabecera Turbo Vision

#include <String.H> // Para usar funciones de cadena

#include <Stdio.H> // Para usar la funcion sprintf()

const cmNueva = 201,
    cmCascada = 202,
    cmMosaico = 203 ;

        //
        // Se deriva una clase a partir de TApplication,
        // a la que se llama AplicacionEstandar
        //

class AplicacionEstandar : public TApplication
{
        public :

                        // Se define el constructor de la clase

                AplicacionEstandar() ;

                static TStatusLine * initStatusLine( TRect ) ; // Línea de estado

                static TMenuBar * initMenuBar( TRect ) ; // Creación del menú

                        // Método para la gestión de eventos

                virtual void handleEvent( TEvent & ) ;

                        // Métodos de respuesta a eventos

                void mNueva() ;

                void mCascada() ;

                void mMosaico() ;

} ;

        // Se crea un nuevo tipo de vista a la que se llama Contenido,
        // derivándola de TScroller, y no de TView

class Contenido : public TScroller
```

```
{
        public :

                        // El constructor recibe además del área de la vista
                        // dos punteros a dos barras de desplazamiento

                Contenido( TRect &, TScrollBar *, TScrollBar * ) ;

                void draw() ; // Método encargado de dibujar la vista
} ;

        // Se implementa el constructor de la clase

AplicacionEstandar::AplicacionEstandar() :
                        TProgInit( & initStatusLine,
                                & initMenuBar,
                                & initDeskTop )

{
}

        // Implementación del método initStatusLine

TStatusLine * AplicacionEstandar::initStatusLine( TRect Area )
{
                // Se define como área de línea de estado la última línea
                // del área total

        Area . a . y = Area . b . y - 1 ;

        return new TStatusLine( Area,
                                * new TStatusDef( 0, 65535 ) +
                                * new TStatusItem( "~Alt-X~ Salir", kbAltX, cmQuit ) +
                                * new TStatusItem( "~F3~ Abrir ventana", kbF3, cmNueva ) +
                                * new TStatusItem( "~F6~ Siguiente", kbF6, cmNext ) +
                                * new TStatusItem( "~F5~ Zoom", kbF5, cmZoom ) +
                                * new TStatusItem( "~Alt-F3~ Cerrar", kbAltF3, cmClose )
                        ) ;
}

        // Implementación del método initMenuBar

TMenuBar * AplicacionEstandar::initMenuBar( TRect Area )
{
        Area . b . y = Area . a . y + 1 ;

        return new TMenuBar( Area,
                        * new TSubMenu( "~V~entanas", kbAltR ) +
                                * new TMenuItem( "~N~ueva", cmNueva, 0, hcNoContext, "" ) +
                                * new TMenuItem( "~C~ascada", cmCascada, 0, hcNoContext, "" ) +
                                * new TMenuItem( "~M~osaico", cmMosaico, 0, hcNoContext, "" )
                        ) ;
}

        // Implementación del método handleEvent

void AplicacionEstandar::handleEvent( TEvent & Evento )
{
                // Primero se ha de procesar el evento por parte
                // del gestor de eventos de nuestra clase base

        TApplication::handleEvent( Evento ) ;

        if( Evento . what == evCommand )        // Si el evento es un comando
        {
                switch( Evento . message . command ) // Según el comando que sea
                {
                        case cmNueva :

                                mNueva() ;      // Llama al método adecuado
                                break ;

                        case cmCascada :

                                mCascada() ;
                                break ;

                        case cmMosaico :

                                mMosaico() ;
                                break ;

                        default :   // Si no es ninguno de los comandos reconocidos

                                return ; // Regresa sin procesarlo
                }

                clearEvent( Evento ) ;   // Si se ha procesado hay que borrarlo
        }
}


        // Implementación del método de respuesta al comando cmNueva

void AplicacionEstandar::mNueva()
{
        static ushort nVentana = 1 ;
```

```
                // Crear una ventana

        TWindow * Ventana = new TWindow( TRect( 5, 5, 35, 15 ), "Ventanas", nVentana++ ) ;

                // Permitir el poner la ventana en forma de cascada o mosaico

        Ventana -> options |= ofTileable ;

                // Se obtiene la extensión actual de la ventana

        TRect Extension = Ventana -> getExtent() ;

                // Se reduce para que la vista que se va a insertar
                // ocupe todo excepto el marco delimitador

        Extension . grow( -1, -1 ) ;

                // Se crea un contenido y se inserta

        Ventana -> insert( new Contenido( Extension,
                                0, // No hay barra de desplazamiento horizontal
                                Ventana -> standardScrollBar( sbVertical | sbHandleKeyboard )
                                ) ) ;

        deskTop -> insert( Ventana ) ; // Insertarla en el despacho
}

        // Implementación del método de respuesta al comando cmCascada

void AplicacionEstandar::mCascada()
{
        deskTop -> cascade( deskTop -> getExtent() ) ;
}

        // Implementación del método de respuesta al comando cmMosaico

void AplicacionEstandar::mMosaico()
{
        deskTop -> tile( deskTop -> getExtent() ) ;
}

        // Implementación del constructor de la clase Contenido

Contenido::Contenido( TRect & Limites, TScrollBar * Hor, TScrollBar * Ver ) :

                        TScroller( Limites, Hor, Ver )
{
                // Se activan los indicadores gfGrowHiX y gfGrowHiY,
                // para que al redimensionar la ventana automáticamente
                // se redimensione el tamaño de Contenido

        growMode = gfGrowHiX | gfGrowHiY ;

                // Se fijan los límites de datos

        setLimit( maxViewWidth, 64 ) ; // un máximo de 64 líneas
}

        // Implementación del método draw

void Contenido::draw()
{
        ushort Color = getColor( 1 ) ; // Se obtiene el color de texto normal

        TDrawBuffer Buffer ; // Buffer para preparar la salida

        int Codigo = delta . y * ( size . x / 7 ) + 1 ; // Contador de códigos ASCII

                // Recorrer todas las líneas que tiene la ventana

        for( int i = 0 ; i < size . y ; i++ )
        {
                int Columnas = size . x / 7 ; // Columnas de códigos posibles

                char Cadena[ 256 ] ; // Para formatear la salida

                strcpy( Cadena, "" ) ; // En principio la cadena está vacía

                        // Hasta llenar la ventana o llegar al código 127

                for( int j = 0 ; j < Columnas && Codigo < 128 ; j++ )
                {
                        char Buffer[ 10 ] ;

                                // Formatear la salida

                        sprintf( Buffer, "%03d %c  ", Codigo++, Codigo ) ;

                        strcat( Cadena, Buffer ) ; // Y añadirla a la Cadena
                }

                        // Limpiar el contenido del buffer

                Buffer . moveChar( 0, ' ', Color, size . x ) ;

                        // Añadirle la cadena a mostrar
```

```
                Buffer . moveStr( 0, Cadena, Color ) ;

                        // Escribir el buffer en la vista

                writeLine( 0, i, size . x, 1, Buffer ) ;
        }
}

int main()
{
        AplicacionEstandar      MiAplicacion ; // Se crea un objeto

        MiAplicacion . run() ; // Y se ejecuta

        return 0 ; // Fin del programa
}
```