Introducción

Requisitos previos

Para el presente curso supondré que estás familiarizado con la programación en C y C++ y también con las aplicaciones y el entorno Windows, al menos al nivel de usuario. Sin embargo, no se requerirán muchos más conocimientos.
El curso pretende ser una explicación de la forma en que se realizan los programas en Windows usando el API. Las explicaciones de las funciones y los mensajes del API son meras traducciones del fichero de ayuda de WIN32 de Microsoft, y sólo se incluyen como complemento.
Para empezar, vamos a ponernos en antecedentes. Veamos primero algunas características especiales de la programación en Windows.

Independencia de la máquina

Los programas Windows son independientes de la máquina en la que se ejecutan (o al menos deberían serlo), el acceso a los dispositivos físicos se hace a través de interfaces, y nunca se accede directamente a ellos. Esta es una de las principales ventajas para el programador, ya que no hay que preocuparse por el modelo de tarjeta gráfica o de impresora, la aplicación funcionará con todas, y será el sistema operativo el que se encargue de que así sea.

Recursos

Un concepto importante es el de recurso. Desde el punto de vista de Windows, un recurso es todo aquello que puede ser usado por una o varias aplicaciones. Existen recursos físicos, que son los dispositivos que componen el ordenador, como la memoria, la impresora, el teclado o el ratón y recursos virtuales o lógicos, como los gráficos, los iconos o las cadenas de caracteres.
Por ejemplo, si nuestra aplicación requiere el uso de un puerto serie, primero debe averiguar si está disponible, es decir, si existe y si no lo está usando otra aplicación; y después lo reservará para su uso. Esto es necesario porque este tipo de recurso no puede ser compartido.
Lo mismo pasa con la memoria o con la tarjeta de sonido, aunque son casos diferentes. Por ejemplo, la memoria puede ser compartida, pero de una forma general, cada porción de memoria no puede compartirse, (al menos en los casos

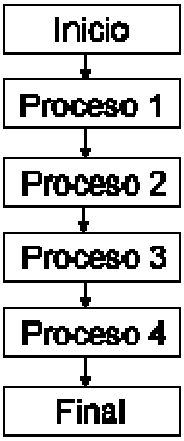
normales, veremos que es posible hacer aplicaciones con memoria compartida), y se trata de un recurso finito. Las tarjetas de sonido, dependiendo del modelo, podrán o no compartirse por varias aplicaciones. Otros recursos como el ratón y el teclado también se comparten, pero se asigna su uso automáticamente a la aplicación activa, a la que normalmente nos referiremos como la que tiene el "foco", es decir, la que mantiene contacto con el usuario.
Desde nuestro punto de vista, como programadores, también consideramos recursos varios componentes como los menús, los iconos, los cuadros de diálogo, las cadenas de caracteres, los mapas de bits, los cursores, etc. En sus programas, el Windows almacena separados el código y los recursos, dentro del mismo fichero, y estos últimos pueden ser editados por separado, permitiendo por ejemplo, hacer versiones de los programas en distintos idiomas sin tener acceso a los ficheros fuente de la aplicación.

Ventanas

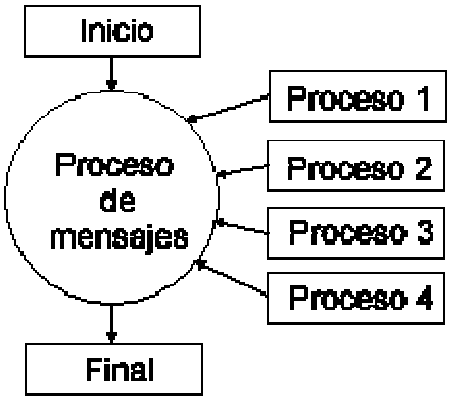
La forma en que se presentan las aplicaciones Windows (al menos las interactivas) ante el usuario, es la ventana. Supongo que todos sabemos qué es una ventana: un área rectangular de la pantalla que se usa de interfaz entre la aplicación y el usuario.
Cada aplicación tiene al menos una ventana, la ventana principal, y todas las comunicaciones entre usuario y aplicación se canalizan a través de una ventana. Cada ventana comparte el espacio de la pantalla con otras ventanas, incluso de otras aplicaciones, aunque sólo una puede estar activa, es decir, sólo una puede recibir información del usuario.

Eventos

Los programas en Windows están orientados a eventos, esto significa que normalmente los programas están esperando a que se produzca un acontecimiento que les incumba, y mientras tanto permanecen aletargados o dormidos.
Un evento puede ser por ejemplo, el movimiento del ratón, la activación de un menú, la llegada de información desde el puerto serie, una pulsación de una tecla...
Esto es así porque Windows es un sistema operativo multitarea, y el tiempo del microprocesador ha de repartirse entre todos los programas que se estén ejecutando. Si los programas fueran secuenciales puros, esto no sería posible, ya que hasta que una aplicación finalizara, el sistema no podría atender al resto.



Ejemplo de programa secuencial:



Ejemplo de programa por eventos:

Proyectos

Debido a la complejidad de los programas Windows, normalmente los dividiremos en varios ficheros fuente, que compilaremos por separado y enlazaremos juntos.
Cada compilador puede tener diferencias, más o menos grandes, a la hora de trabajar con proyectos. Sin embargo creo que no deberías tener grandes dificultades para adaptarte a cada uno de ellos.
En el presente curso trabajaremos con el compilador de "Bloodshed", que es público y gratuito, y puede descargarse de Internet en la siguiente URL: http://www.bloodshed.net/.
Para crear un proyecto Windows usando este compilador elegiremos el menú "File/New Project". Se abrirá un cuadro de diálogo donde podremos elegir el tipo de proyecto. Elegiremos "Windows Application" y "C++ Project". A continuación pulsamos "Aceptar".
El compilador crea un proyecto con un fichero C++, con el esqueleto de una aplicación para una ventana, a partir de ahí empieza nuestro trabajo.

Convenciones

En parte para que no te resulte muy difícil adaptarte a la terminología de Windows, y a la documentación existente, y en parte para seguir mi propia costumbre, en la mayoría de los casos me referiré a componentes y propiedades de Windows con sus nombres en inglés. Por ejemplo, hablaremos de "button", "check box", "radio button", "list box", "combo box" o "property sheet", aunque algunas veces traduzca sus nombre a español, por ejemplo, "dialog box" se nombrará a menudo como "cuadro de diálogo".
Además hablaremos a menudo de ventanas "overlapped" o superponibles, que son las ventanas corrientes. Para el término "pop-up" he desistido de buscar una traducción.
También se usaran a menudo, con relación a "check boxes", términos ingleses como checked, unchecked o grayed, en lugar de marcado, no marcado o gris.
Owner-draw, es un estilo que indica que una ventana o control no es la encargada de actualizarse en pantalla, esa responsabilidad es transferida a la ventana dueña del control o ventana.
Para "bitmap" se usará a menudo la expresión "mapa de bits".

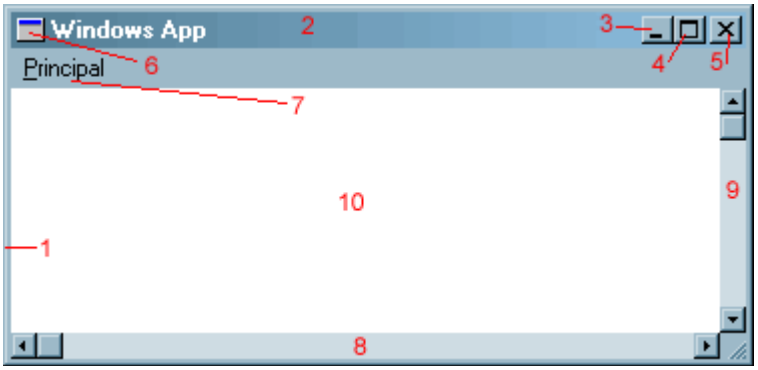
Controles

Los controles son la forma en que las aplicaciones Windows intercambian datos con el usuario. Normalmente se usan dentro de los cuadros de diálogo, pero en realidad pueden usarse en cualquier ventana.
Existen bastantes, y los iremos viendo poco a poco, al mismo tiempo que aprendemos a manejarlos.
Los más importantes y conocidos son:

- control estatic: son etiquetas, marcos, iconos o dibujos.
- control edit: permiten que el usuario introduzca datos alfanuméricos en la aplicación.
- control list box: el usuario puede escoger entre varias opciones de una lista.
- control combo box: es una combinación entre un edit y un list box.
- control scroll bar: barras de desplazamiento, para la introducción de valores entre márgenes definidos.
- control button: realizan acciones o comandos, de button de derivan otros dos controles muy comunes:
- control check box: permite leer variables de dos estados "checked" o "unchecked"
- control radio button: se usa en grupos, dentro de cada grupo sólo uno puede ser activado.

Capítulo 1 Componentes de una ventana

Veamos ahora los elementos que componen una ventana, aunque más adelante veremos que no todos tienen por qué estar presentes en todas las ventanas.



El borde de la ventana

Hay varios tipos, dependiendo de que estén o no activas las opciones de cambiar el tamaño de la ventana. Se trata de un área estrecha alrededor de la ventana que permite cambiar su tamaño (1).

Barra de título

Zona en la parte superior de la ventana que contiene el icono y el título de la ventana, esta zona también se usa para mover la ventana a través de la pantalla, y mediante doble clic, para cambiar entre el modo maximizado y tamaño normal (2).

Caja de minimizar

Pequeña área cuadrada situada en la parte derecha de la barra de título que sirve para disminuir el tamaño de la ventana. Antes de la aparición del Windows 95 la ventana se convertía a su forma icónica, pero desde la aparición del Windows 95 los iconos desaparecieron, la ventana se oculta y sólo permanece un botón en la barra de estado (3).

Caja de maximizar

Pequeña área cuadrada situada en la parte derecha de la barra de título que sirve para agrandar la ventana para que ocupe toda la pantalla. Cuando la ventana está maximizada, se sustituye por la caja de restaurar (4).

Caja de cerrar

Pequeña área cuadrada situada en la parte derecha de la barra de título que sirve para cerrar la ventana. (5)

Caja de control de menú

Pequeña área cuadrada situada en la parte izquierda de la barra de título, normalmente contiene el icono de la ventana, y sirve para desplegar el menú del sistema (6).

Menú

O menú del sistema. Se trata de una ventana especial que contiene las funciones comunes a todas las ventanas, también accesibles desde las cajas y el borde, como minimizar, restaurar, maximizar, mover, cambiar tamaño y cerrar. Este menú se despliega al pulsar sobre la caja de control de menú.

Barra de menú

Zona situada debajo de la barra de título, contiene los menús de la aplicación (7).

Barra de scroll horizontal

Barra situada en la parte inferior de la ventana, permite desplazar horizontalmente la vista del área de cliente (8).

Barra de scroll vertical

Barra situada en la parte derecha de la ventana, permite desplazar verticalmente la vista del área de cliente (9).

El área de cliente

Es la zona donde el programador sitúa los controles, y los datos para el usuario. En general es toda la superficie de la ventana lo que no está ocupada por las zonas anteriores (10).

Capítulo 2 Notación Húngara

La notación húngara es un sistema usado normalmente para crear los nombres de variables cuando se programa en Windows. Es el sistema usado en la programación del sistema operativo, y también por la mayoría de los programadores. A veces también usaremos este sistema en algunos ejemplos de este curso.
Consiste en prefijos en minúsculas que se añaden a los nombres de las variables, y que indican su tipo. El resto del nombre indica, lo más claramente posible, la función que realiza la variable.

Prefijo	Significado
b	Booleano
c	Carácter (un byte)
dw	Entero largo de 32 bits sin signo (DOBLE WORD)
f	Flags empaquetados en un entero de 16 bits
h	Manipulador de 16 bits (HANDLE)
l	Entero largo de 32 bits
lp	Puntero a entero largo de 32 bits
lpfn	Puntero largo a una función que devuelve un entero
lpsz	Puntero largo a una cadena terminada con cero
n	Entero de 16 bits
p	Puntero a entero de 16 bits
pt	Coordenadas (x, y) empaquetadas en un entero de 32 bits
rgb	Valor de color RGB empaquetado en un entero de 32 bits
sz	Cadena terminada en cero
w	Entero corto de 16 bits sin signo (WORD)

Ejemplos

nContador: la variable es un entero que se usará como contador.
szNombre: una cadena terminada con cero que almacena un nombre.
bRespuesta: una variable booleana que almacena una respuesta.

Capítulo 3 Estructura de un programa Windows GUI

Hay algunas diferencias entre la estructura de un programa C/C++ normal, y la correspondiente a un programa Windows GUI. Algunas de estas diferencias se deben a que los programas GUI estás basados en mensajes, otros son sencillamente debidos a que siempre hay un determinado número de tareas que hay que realizar.

```
// Ficheros include:
#include <windows.h>

// Prototipos:
LRESULT   CALLBACK   WindowProcedure   (HWND,   UINT,
WPARAM, LPARAM);

// Función de entrada:
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszCmdParam,
                    int nCmdShow)
{
   // Declaración:
   // Inicialización:
   // Bucle de mensajes:
   return Message.wParam;
}

// Definición de funciones:
```

Cabeceras

Lo primero es lo primero, para poder usar las funciones del API de Windows hay que incluir al menos un fichero de cabecera, pero generalmente no bastará con uno.
El fichero <windows.h> lo que hace es incluir la mayoría de los ficheros de cabecera corrientes en aplicaciones GUI, pero podemos incluir sólo los que necesitemos, siempre que sepamos cuales son. Por ejemplo, la función WinMain está declarada en el fichero de cabecera *winbase.h*.
Generalmente esto resultará incómodo, ya que para cada nueva función, mensaje o estructura tendremos que comprobar, y si es necesario, incluir nuevos ficheros. Es mejor usar *windows.h* directamente.

Prototipos

Cada tipo (o clase) de ventana que usemos en nuestro programa (normalmente sólo será una), o cada cuadro de diálogo (de estos puede haber muchos), necesitará un procedimiento propio, que deberemos declarar y definir. Siguiendo la estructura de un programa C, esta es la zona normal de declaración de prototipos.

Función de entrada, WinMain

La función de entrada de un programa Windows es "WinMain", en lugar de la conocida "main". Normalmente, la definición de esta función cambia muy poco de una aplicaciones a otras. Se divide en tres partes claramente diferenciadas: declaración, inicialización y bucle de mensajes.

Parámetros de entrada de "WinMain"

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow)
```

La función WinMain tiene cuatro parámetros de entrada:

- *hInstance* es un manipulador para la instancia del programa que estamos ejecutando. Cada vez que se ejecuta una aplicación, Windows crea una Instancia para ella, y le pasa un manipulador de dicha instancia a la aplicación.
- *hPrevInstance* es un manipulador a instancias previas de la misma aplicación. Como Windows es multitarea, pueden existir varias versiones de la misma aplicación ejecutándose, varias instancias. En Windows 3.1, este parámetro nos servía para saber si nuestra aplicación ya se estaba ejecutando, y de ese modo se podían compartir los datos comunes a todas las instancias. Pero eso era antes, ya que en Win32 usa un segmento distinto para cada instancia y este parámetro es siempre NULL, sólo se conserva por motivos de compatibilidad.
- *lpszCmdParam*, esta cadena contiene los argumentos de entrada del comando de línea.
- *nCmdShow*, este parámetro especifica cómo se mostrará la ventana. Para ver sus posibles valores consultar valores de ncmdshow. Se recomienda no usar este parámetro en la función ShowWindow la primera vez que se ésta es llamada. En su lugar debe usarse el valor SW_SHOWDEFAULT.

Función WinMain típica

```
int  WINAPI  WinMain(HINSTANCE  hInstance,  HINSTANCE
hPrevInstance,
   LPSTR lpszCmdParam, int nCmdShow)
{
   /* Declaración: */
   HWND hwnd;
   MSG mensaje;
   WNDCLASSEX wincl;

   /* Inicialización: */
   /* Estructura de la ventana */
   wincl.hInstance = hInstance;
   wincl.lpszClassName = "NUESTRA_CLASE";
   wincl.lpfnWndProc = WindowProcedure;
   wincl.style = CS_DBLCLKS;
   wincl.cbSize = sizeof (WNDCLASSEX);

   /* Usar icono y puntero por defecto */
   wincl.hIcon = LoadIcon (NULL, IDI_APPLICATION);
   wincl.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
   wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
   wincl.lpszMenuName = NULL;
   wincl.cbClsExtra = 0;
   wincl.cbWndExtra = 0;
   wincl.hbrBackground = (HBRUSH) COLOR_BACKGROUND;

   /* Registrar la clase de ventana, si falla, salir
del programa */
   if(!RegisterClassEx(&wincl)) return 0;

   hwnd = CreateWindowEx(
           0,
           "NUESTRA_CLASE",
           "Ejemplo 001",
           WS_OVERLAPPEDWINDOW,
           CW_USEDEFAULT,
           CW_USEDEFAULT,
           544,
           375,
```

```
            HWND_DESKTOP,
            NULL,
            hThisInstance,
            NULL
    );

    ShowWindow(hwnd, SW_SHOWDEFAULT);

    /* Bucle de mensajes: */
    while(TRUE == GetMessage(&mensaje, 0, 0, 0))
    {
       TranslateMessage(&mensaje);
       DispatchMessage(&mensaje);
    }

    return mensaje.wParam;
}
```

Declaración

En la primera zona declararemos las variables que necesitamos para nuestra función WinMain, que como mínimo serán tres:

- *HWND hWnd*, un manipulador para la ventana principal de la aplicación. Ya sabemos que nuestra aplicación necesitará al menos una ventana.
- *MSG Message*, una variable para manipular los mensajes que lleguen a nuestra aplicación.
- *WNDCLASSEX wincl*, una estructura que se usará para registrar la clase particular de ventana que usaremos en nuestra aplicación. Existe otra estructura para registrar clases que se usaba antiguamente, pero que ha sido desplazada por esta nueva versión, se trata de WNDCLASS.

Inicialización

Esta zona se encarga de registrar la clase o clases de ventana, crear la ventana y visualizarla en pantalla.

Para registrar la clase primero hay que rellenar adecuadamente la estructura WNDCLASSEX, que define algunas características que serán comunes a todas las ventanas de una misma clase, como color de fondo, icono, menú por defecto, el procedimiento de ventana, etc. Después hay que llamar a la función RegisterClassEx.

En el caso de usar una estructura WNDCLASS se debe registrar la clase usando la función RegisterClass.

A continuación se crea la ventana usando la función CreateWindowEx, la función CreateWindow ha caído prácticamente en desuso. Cualquiera de estas dos funciones nos devuelve un manipulador de ventana que podemos necesitar en otras funciones, sin ir más lejos, la siguiente.
Pero esto no muestra la ventana en la pantalla. Para que la ventana sea visible hay que llamar a la función ShowWindow. La primera vez que se llama a ésta función, después de crear la ventana, se puede usar el parámetro nCmdShow de WinMain como parámetro o mejor aún, como se recomienda por Windows, el valor SW_SHOWDEFAULT.

Bucle de mensajes

Este es el núcleo de la aplicación, como se ve en el ejemplo el programa permanece en este bucle mientras la función GetMessage retorne con un valor TRUE.

```
while(TRUE == GetMessage(&mensaje, 0, 0, 0)) {
   TranslateMessage(&mensaje);
   DispatchMessage(&mensaje);
}
```

Este es el bucle de mensajes recomendable, aunque no sea el que se usa habitualmente. La razón es que la función GetMessage puede retornar tres valores: TRUE, FALSE ó -1. El valor -1 indica un error, así que en este caso se debería abandonar el bucle.
El bucle de mensajes que encontraremos habitualmente es este:

```
while(GetMessage(&mensajee, 0, 0, 0)) {
   TranslateMessage(&mensaje);
   DispatchMessage(&mensaje);
}
```

NOTA: El problema con este bucle es que si GetMessage regresa con un valor -1, que indica un error, la condición del "while" se considera verdadera, y el bucle continúa. Si el error es permanente, el programa jamás terminará.

La función TranslateMessage se usa para traducir los mensajes de teclas virtuales a mensajes de carácter. Veremos esto con más detalle en el capítulo dedicado al teclado (cap. 34).
Los mensajes traducidos se reenvían a la lista de mensajes del proceso, y se recuperarán con las siguientes llamadas a GetMessage.
La función DispatchMessage envía el mensaje al procedimiento de ventana, donde será tratado adecuadamente. El próximo capítulo está dedicado al

procedimiento de ventana, y al final de él estaremos en disposición de crear nuestro primer programa Windows.

Definición de funciones

En esta parte definiremos, entre otras cosas, los procedimientos de ventana, que se encargan de procesar los mensajes que lleguen a cada ventana.

Capítulo 4 El procedimiento de ventana

Cada ventana tiene una función asociada, esta función se conoce como procedimiento de ventana, y es la encargada de procesar adecuadamente todos los mensajes enviados a una determinada clase de ventana. Es la responsable de todo lo relativo al aspecto y al comportamiento de una ventana.
Normalmente, estas funciones están basadas en una estructura "switch" donde cada "case" corresponde aun determinado tipo de mensaje.

Sintaxis

```
LRESULT CALLBACK WindowProcedure(
   HWND hwnd,      // Manipulador de ventana
   UINT msg,       // Mensaje
   WPARAM wParam, // Parámetro palabra, varía
   LPARAM lParam  // Parámetro doble palabra, varía
  );
```

- hwnd es el manipulador de la ventana a la que está destinado el mensaje.
- msg es el código del mensaje.
- wParam es el parámetro de tipo palabra asociado al mensaje.
- lParam es el parámetro de tipo doble palabra asociado al mensaje.
-

Podemos considerar este prototipo como una *plantilla* para crear nuestros propios procedimientos de ventana. El nombre de la función puede cambiar, pero el valor de retorno y los parámetros deben ser los mismos. El miembro *lpfnWndProc* de la estructura WNDCLASS es un puntero a una función de este tipo, esa función es la que se encargará de procesar todos los mensajes para esa clase de ventana. Cuando registremos nuestra clase de ventana, tendremos que asignar a ese miembro el puntero a nuestro procedimiento de ventana.
Para más detalles sobre la función de procedimiento de ventana, consultar WindowProc.

Prototipo de procedimiento de ventana

```
LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM,
LPARAM);
```

Implementación de procedimiento de ventana simple

```
/* Esta función es llamada por la función del API
DispatchMessage() */
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT
msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)                  /* manipulador del
mensaje */
    {
        case WM_DESTROY:
           PostQuitMessage(0);        /* envía un
mensaje WM_QUIT a la cola de mensajes */
           break;
        default:                        /* para los
mensajes de los que no nos ocupamos */
           return DefWindowProc(hwnd, msg, wParam,
lParam);
    }
    return 0;
}
```

En general, habrá tantos procedimientos de ventana como programas diferentes y todos serán distintos, pero también tendrán algo en común: todos ellos procesarán los mensajes que lleguen a una clase de ventana.

En este ejemplo sólo procesamos un tipo de mensaje, se trata de WM_DESTROY que es el mensaje que se envía a una ventana cuando se recibe un comando de cerrar, ya sea por menú o mediante el icono de aspa en la esquina superior derecha de la ventana.

Este mensaje sólo sirve para informar a la aplicación de que el usuario tiene la intención de abandonar la aplicación, y le da una oportunidad de dejar las cosas en su sitio: cerrar ficheros, liberar memoria, guardar variables, etc. Incluso, la aplicación puede decidir que aún no es el momento adecuado para abandonar la aplicación. En el caso del ejemplo, efectivamente cierra la aplicación, y lo hace enviándole un mensaje WM_QUIT, mediante la función PostQuitMessage.

El resto de los mensajes se procesan en el caso "default", y simplemente se cede su tratamiento a la función del API que hace el proceso por defecto para cada mensaje, DefWindowProc.
Este es el camino que sigue el mensaje WM_QUIT cuando llega, ya que el proceso por defecto para este mensaje es cerrar la aplicación.
En posteriores capítulos veremos como se complica paulatinamente esta función, añadiendo más y más mensajes.

Capítulo 5 Menús 1

Ahora que ya sabemos hacer el esqueleto de una aplicación Windows, veamos el primer medio para comunicarnos con ella.
Supongo que todos sabemos lo que es un menú: se trata de una ventana un tanto especial, del tipo pop-up, que contiene una lista de comandos u opciones entre las cuales el usuario puede elegir.
Cuando se usan en una aplicación, normalmente se agrupan varios menús bajo una barra horizontal, (que no es otra cosa que un menú), dividida en varias zonas o ítems.
Cada ítem de un menú, (salvo los separadores y aquellos que despliegan nuevos menús), tiene asociado un identificador. El valor de ese identificador se usará por la aplicación para saber qué opción se activó por el usuario, y decidir las acciones a tomar en consecuencia.
Existen varias formas de añadir un menú a una ventana, veremos cada una de ellas por separado.
También es posible desactivar o inhibir algunas opciones para que no estén disponibles para el usuario.

Usando las funciones para inserción ítem a ítem

Este es el sistema más rudimentario, pero como ya veremos en el futuro, en ocasiones puede ser muy útil. Empezaremos viendo este sistema porque ilustra mucho mejor la estructura de los menús.
Tomemos el ejemplo del capítulo anterior y definamos algunas constantes:

```
#define CM_PRUEBA 100
#define CM_SALIR  101
```

Y añadamos la declaración de una función en la zona de prototipos:

```
void InsertarMenu(HWND);
```

Al final del programa añadimos la definición de esta función:

```
void InsertarMenu(HWND hWnd)
{
   HMENU hMenu1, hMenu2;

   hMenu1 = CreateMenu(); /* Manipulador de la barra
de menú */
   hMenu2 = CreateMenu(); /* Manipulador para el
primer menú pop-up */
   AppendMenu(hMenu2,      MF_STRING,      CM_PRUEBA,
"&Prueba"); /* 1º ítem */
   AppendMenu(hMenu2,    MF_SEPARATOR,    0,    NULL);
/* 2º ítem (separador) */
   AppendMenu(hMenu2,      MF_STRING,      CM_SALIR,
"&Salir");   /* 3º ítem */
   /* Inserción del menú pop-up */
   AppendMenu(hMenu1,    MF_STRING    |    MF_POPUP,
(UINT)hMenu2, "&Principal");
   SetMenu (hWnd, hMenu1);  /* Asigna el menú a la
ventana hWnd */
}
```

Y por último, sólo nos queda llamar a nuestra función, insertaremos ésta llamada justo antes de visualizar la ventana.

```
...
   InsertarMenu(hWnd);
   ShowWindow(hWnd, SW_SHOWDEFAULT);
...
```

Veamos cómo funciona "InsertarMenu".
La primera novedad son las variables del tipo HMENU. HMENU es un tipo de manipulador especial para menús. Necesitamos dos variables de este tipo, una

para manipular la barra de menú, hMenu1. La otra para manipular cada uno de los menús pop-up, en este caso sólo uno, hMenu2.
De momento haremos una barra de menú con un único elemento que será un menú pop-up. Después veremos como implementar menús más complejos.
Para crear un menú usaremos la función CreateMenu, que crea un menú vacío.
Para ir añadiendo ítems a cada menú usaremos la función AppendMenu. Esta función tiene varios argumentos:
El primero es el menú donde queremos insertar el nuevo ítem.
El segundo son las opciones o atributos del nuevo ítem, por ejemplo MF_STRING, indica que se trata de un ítem de tipo texto, MF_SEPARATOR, es un ítem separador y MF_POPUP, indica que se trata de un menú que desplegará un nuevo menú pop-up.
El siguiente parámetro puede tener distintos significados:

- Puede ser un identificador de comando, este identificador se usará para comunicar a la aplicación si el usuario selecionó un determinado ítem.
- Un manipulador de menú, si el ítem tiene el flag MF_POPUP, en este caso hay que hacer un casting a (UINT).
- O también puede ser cero, si se trata de un separador.

El último parámetro es el texto del ítem, cuando se ha especificado el flag MF_STRING, más adelante veremos que los ítems pueden ser también bitmaps. Normalmente se trata de una cadena de texto. Pero hay una peculiaridad interesante, para indicar la tecla que activa un determinado ítem de un menú se muestra la letra correspondiente subrayada. Esto se consigue insertando un '&' justo antes de la letra que se quiere usar como atajo, por ejemplo, en el ítem "&Prueba" esta letra será la 'P'.
Por último SetMenu, asigna un menú a una ventana determinada. El primer parámetro es el manipulador de la ventana, y el segundo el del menú.
Prueba estas funciones y juega un rato con ellas. A continuación veremos cómo hacer que nuestra aplicación responda a los mensajes del menú.

Uso básico de MessageBox

Antes de aprender a visualizar texto en la ventana, usaremos un mecanismo más simple para informar al usuario de cualquier cosa que pase en nuestra aplicación. Este mecanismo no es otro que el cuadro de mensaje (message box), que consiste en una pequeña ventana con un mensaje para el usuario y uno o varios botones, según el tipo de cuadro de mensaje que usemos. En nuestros primeros ejemplos, el cuadro de mensaje sólo incluirá el botón de "Aceptar".
Para visualizar un cuadro de mensaje simple, usaremos la función MessageBox. En nuestros ejemplos bastará con la siguiente forma:

```
MessageBox(hWnd, "Texto de mensaje", "Texto de título", MB_OK);
```

Esto mostrará un pequeño cuadro de diálogo con el texto y el título especificados y un botón de "Aceptar". El cuadro se cerrará al pulsar el botón o al pulsar la tecla de Retorno.

Respondiendo a los mensajes del menú

Las activaciones de los menús se reciben mediante un mensaje WM_COMMAND. Para procesar estos mensajes, si sólo podemos recibir mensajes desde un menú, únicamente nos interesa la palabra de menor peso del parámetro wParam del mensaje.

Modifiquemos el procedimiento de ventana para procesar los mensajes de nuestro menú:

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT
msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)                  /* manipulador del
mensaje */
    {
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case CM_PRUEBA:
                 MessageBox(hwnd, "Comando: Prueba",
"Mensaje de menú", MB_OK);
                 break;
              case CM_SALIR:
                 MessageBox(hwnd, "Comando: Salir",
"Mensaje de menú", MB_OK);
                 /* envía un mensaje WM_QUIT a la
cola de mensajes */
                 PostQuitMessage(0);
                 break;
           }
           break;
        case WM_DESTROY:
           /* envía un mensaje WM_QUIT a la cola de
mensajes */
           PostQuitMessage(0);
           break;
```

```
        default: /* para los mensajes de los que no
nos ocupamos */
            return DefWindowProc(hwnd, msg, wParam,
lParam);
    }
    return 0;
}
```

Sencillo, ¿no?.
Observa que hemos usado la macro LOWORD para extraer el identificador del ítem del parámetro wParam. Después de eso, todo es más fácil.
También se puede ver que hemos usado la misma función para salir de la aplicación que para el mensaje WM_DESTROY: la función PostQuitMessage.

Ficheros de recursos

Veamos ahora una forma más sencilla y más frecuente de implementar menús.
Lo normal es implementar los menús desde un fichero de recursos, el sistema que hemos visto sólo se usa en algunas ocasiones, para crear o modificar menús durante la ejecución de la aplicación.
Es importante adquirir algunas buenas costumbres cuando se trabaja con ficheros de recursos.

1. Usaremos siempre etiquetas como identificadores para los ítems de los menús, y nunca valores numéricos literales.
2. Crearemos un fichero de cabecera con las definiciones de los identificadores, en nuestro ejemplo se llamará "ids.h".
3. Incluiremos este fichero de cabecera tanto en el fichero de recursos y como en el del código fuente de nuestra aplicación.

Partimos de un proyecto nuevo: win003. Pero usaremos el código modificado del ejemplo1.
Para ello creamos un nuevo proyecto de tipo GUI, al que llamaremos Win003, y copiamos el contenido de "ejemplo1.c" en el fichero "main.cpp", al que renombraremos como "win003.c".
A continuación crearemos el fichero de identificadores.
Añadimos el fichero de cabecera a nuestro proyecto. Si estás usando Dev-C++, ésto se hace pulsando con el botón derecho del ratón sobre el nodo del proyecto y eligiendo el ítem de "Buevo código fuente" en el menú que se despliega.
Después lo renombramos, el mecanismo es similar, pulsamos con el botón derecho sobre el ítem "SiNombre1" y elegimos la opción de "Renombrar archivo" del menú que se despliegue. Como nuevo nombre elegimos: "ids.h".
Introducimos en é los identificadores:

```
#define CM_PRUEBA 100
```

```
#define CM_SALIR 101
```

En el fichero "win003.c" añadimos la línea:

```
#include "ids.h"
```

Justo después de la línea "#include <windows.h>".
Ahora añadiremos el fichero de recursos. Para ello haremos lo mismo que hemos hecho con el fichero "ids.h", pero usaremos el nombre "win003.rc".
En la primera línea introducimos la siguiente línea:

```
#include "ids.h"
```

Y a continuación escribimos:

```
Menu MENU
BEGIN
   POPUP "&Principal"
      BEGIN
         MENUITEM "&Prueba", CM_PRUEBA
         MENUITEM SEPARATOR
         MENUITEM "&Salir", CM_SALIR
      END
END
```

En un fichero de recursos podemos crear toda la estructura de un menú fácilmente. Este ejemplo crea una barra de menú con una columna "Principal", con dos opciones: "Prueba" y "Salir", y con un separador entre ellas.
La sintaxis es sencilla, definimos el menú mediante una cadena identificadora, sin comillas, seguida de la palabra MENU. Entre las palabras *BEGIN* y *END* podemos incluir items, separadores u otras columnas. Para incluir columas usamos una sentencia del tipo POPUP seguida de la cadena que se mostrará como texto en el menú. Cada POPUP se comporta del mismo modo que un MENU.
Los ítems se crean usado la palabra MENUITEM seguida de la cadena que se mostrará en el menú, una coma, y el comando asignado a ese ítem, que puede ser un número entero, o, como en este caso, una macro definida.
Los separadores se crean usando MENUITEM seguido de la palabra *SEPARATOR*.

Observarás que las cadenas que se muestran en el menú contienen un símbolo & en su interior, por ejemplo "&Prueba". Este símbolo indica que la siguiente letra puede usarse para activar la opción del menú desde el teclado, usando la tecla [ALT] más la letra que sigue al símbolo &. Para indicar eso, en pantalla, esa letra se muestra subrayada, en este ejemplo "Prueba".
Ya podemos cerrar el cuadro de edición del fichero de recursos.
Para ver más detalles sobre el uso de este recurso puedes consultar las claves: MENU, POPUP y MENUITEM.

Cómo usar los recursos de menú

Ahora tenemos varias opciones para usar el menú que acabamos de crear.
Primero veremos cómo cargarlo y asignarlo a nuestra ventana, ésta es la forma que más se parece a la del ejemplo del capítulo anterior. Para ello basta con insertar este código antes de llamar a la función ShowWindow:

```
HMENU hMenu;
...
hMenu = LoadMenu(hInstance, "Menu");
SetMenu (hWnd, hMenu);
```

O simplemente:

```
SetMenu (hWnd, LoadMenu(hInstance, "Menu"));
```

La función LoadMenu se encarga de cargar el recurso de menú, para ello hay que proporcionarle un manipulador de la instancia a la que pertenece el recurso y el nombre del menú.
Otro sistema, más sencillo todavía, es asignarlo como menú por defecto de la clase. Para esto basta con la siguiente asignación:

```
WNDCLASSEX wincl;
...
wincl.lpszMenuName = "Menu";
```

Y por último, también podemos asignar un menú cuando creamos la ventana, especificándolo en la llamada a CreateWindowEx:

```
 hwnd = CreateWindowEx(
```

```
            0,
            "NUESTRA_CLASE",
            "Ejemplo 003",
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT,
            CW_USEDEFAULT,
            544,
            375,
            HWND_DESKTOP,
            LoadMenu(hInstance,  "Menu"),  /*  Carga  y
asignación de menú */
            hInstance,
            NULL
    );
```

El tratamiento de los comandos procedentes del menú es igual que en el apartado anterior.

Capítulo 6 Diálogo básico

Los cuadros de diálogo son la forma de ventana más habitual de comunicación entre una aplicación Windows y el usuario. Para facilitar la tarea del usuario a la hora de introducir datos, existen varios tipos de controles, cada uno de ellos diseñado para un tipo específico de información. Los más comunes son los "static", "edit", "button", "listbox", "scroll", "combobox", "group", "checkbutton" y "ratiobutton". A partir de Windows 95 se indrodujeron varios controles nuevos: "updown", "listview", "treeview", "gauge", "tab" y "trackbar".
En realidad, un cuadro de diálogo es una ventana normal, aunque con algunas peculiaridades. También tiene su procedimiento de ventana (procedimiento de diálogo), pero puede devolver un valor a la ventana que lo invoque.
Igual que los menús, los cuadros de diálogo se pueden construir durante la ejecución o a partir de un fichero de recursos.

Ficheros de recursos

La mayoría de los compiladores de C/C++ que incluyen soporte para Windows poseen herramientas para la edición de recursos: menús, diálogos, bitmaps, etc. Sin embargo considero que es interesante que aprendamos a construir nuestros recursos con un editor de textos, cada compilador tiene sus propios editores de recursos, y no tendría sentido explicar cada uno de ellos. El compilador que

usamos "Dev C++", en su versión 4, tiene un editor muy limitado y no aconsejo su uso. De hecho, en la versión actual ya no se incluye, y los ficheros de recursos se editan en modo texto.
De modo que aprenderemos a hacer cuadros de diálogo igual que hemos aprendido a hacer menús: usando el editor de texto.
Para el primer programa de ejemplo de programa con diálogos, que será el ejemplo 4, partiremos de nuevo del programa del ejemplo 1. Nuestro primer diálogo será muy sencillo: un simple cuadro con un texto y un botón de "Aceptar".
Este es el código del fichero de recursos:

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 118, 48
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
 CONTROL "Mensaje de prueba", TEXTO, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 84, 8
 CONTROL "Aceptar", IDOK, "button",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    56, 26, 50, 14
END
```

Necesitamos incluir el fichero "windows.h" ya que en él se definen muchas constantes, como por ejemplo "IDOK" que es el identificador que se usa para el botón de "Aceptar".

También necesitaremos el fichero "ids.h", para definir los identificadores que usaremos en nuestro programa, por ejemplo el identificador del comando de menú para abrir nuestro diálogo.

```
/* Identificadores */

/* Identificadores de comandos */
#define CM_DIALOGO 101
#define TEXTO      100
```

Lo primero que hemos definido es un menú para poder comunicarle a nuestra aplicación que queremos abrir un cuadro de diálogo.
A continuación está la definición del diálogo, que se compone de varias líneas. Puedes ver más detalles en el apartado de recursos dedicado al recurso diálogo.
De momento bastará con un identificador, como el que usábamos para los menús, y además las coordenadas y dimensiones del diálogo.
En cuanto a los estilos, las constantes para definir los estilos de ventana, que comienzan con "WS_", puedes verlos con detalle en la sección de constantes "estilos de ventana". Y los estilos de diálogos, que comienzan con "DS_", en "estilos de diálogo".
Para empezar, hemos definido los siguientes estilos:

- DS_MODALFRAME: indica que se creará un cuadro de diálogo con un marco de dialog-box modal que puede combinarse con una barra de título y un menú de sistema.
- WS_POPUP: crea una ventana "pop-up".
- WS_VISIBLE: crea una ventana inicialmente visible.
- WS_CAPTION: crea una ventana con una barra de título, (incluye el estilo WS_BORDER).

La siguiente línea es la de CAPTION, en ella especificaremos el texto que aparecerá en la barra de título del diálogo.
La línea de FONT sirve para especificar el tamaño y el tipo de fuente de caracteres que usará nuestro diálogo.
Después está la zona de controles, en nuestro ejemplo sólo hemos incluido un texto estático y un botón.
Un control estático (static) nos sirve para mostrar textos o rectángulos, que podemos usar para informar al usuario de algo, como etiquetas o como adorno. Para más detalles ver control static.

```
CONTROL "Mensaje de prueba", -1, "static",
   SS_LEFT | WS_CHILD | WS_VISIBLE,
   8, 9, 84, 8
```

- CONTROL es una palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, introducimos el texto que se mostrará.

- id es el identificador del control. Como los controles static no se suelen manejar por las aplicaciones no necesitamos un identificador, así que ponemos -1.
- class es la clase de control, en nuestro caso "static".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo estático y varios de ventana:
  - SS_LEFT: indica un simple rectángulo y el texto suministrado se alinea en su interior a la izquierda.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

El control button nos sirve para comunicarnos con el diálogo, podemos darle comandos del mismo tipo que los que proporciona un menú. Para más detalles ver recurso button.

```
CONTROL "Aceptar", IDOK, "button",
   BS_PUSHBUTTON | BS_CENTER | WS_CHILD | WS_VISIBLE
| WS_TABSTOP,
   56, 26, 50, 14
```

- CONTROL es una palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, introducimos el texto que se mostrará en su interior.
- id es el identificador del control. Nuestra aplicación recibirá este identificador junto con el mensaje WM_COMMAND cuando el usuario active el botón. La etiqueta IDOK está definida en el fichero Windows.h.
- class es la clase de control, en nuestro caso "button".
- style es el estilo de control que queremos. En nuestro caso es una combinación de varios estilos de button y varios de ventana:
  - BS_PUSHBUTTON: crea un botón corriente que envía un mensaje WM_COMMAND a su ventana padre cuando el usuario selecciona el botón.
  - BS_CENTER: centra el texto horizontalmente en el área del botón.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_TABSTOP: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo WS_TABSTOP.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Procedimiento de diálogo

Como ya hemos dicho, un diálogo es básicamente una ventana, y al igual que aquella, necesita un procedimiento asociado que procese los mensajes que le sean enviados, en este caso, un procedimiento de diálogo.

Sintaxis

```
BOOL CALLBACK DialogProc(
    HWND  hwndDlg,    //  manipulador  del  cuadro  de
diálogo
    UINT uMsg,     // mensaje
    WPARAM wParam, // primer parámetro del mensaje
    LPARAM lParam  // segundo parámetro del mensaje
   );
```

- hwndDlg identifica el cuadro de diálogo y es el manipulador de la ventana a la que está destinado el mensaje.
- msg es el código del mensaje.
- wParam es el parámetro de tipo palabra asociado al mensaje.
- lParam es el parámetro de tipo doble palabra asociado al mensaje.

La diferencia con el procedimiento de ventana que ya hemos visto está en el tipo de valor de retorno, que es el caso del procedimiento de diálogo es de tipo booleano. Puedes consultar una sintaxis más completa de esta función en DialogProc.

Excepto en la respuesta al mensaje WM_INITDIALOG, el procedimiento de diálogo debe retornar con un valor no nulo si procesa el mensaje y cero si no lo hace. Cuando responde a un mensaje **WM_INITDIALOG**, el procedimiento debe retornar cero si llama a la función SetFocus para poner el foco a uno de los controles del cuadro de diálogo. En otro caso, debe retornar un valor distinto de cero, y el sistema pondrá el foco en el primer control del diálogo que pueda recibirlo.

Prototipo de procedimiento de diálogo

El prototipo es parecido al de los procedimientos de ventana:

```
BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);
```

Implementación de procedimiento de diálogo para nuestro ejemplo

Nuestro ejemplo es muy sencillo, ya que nuestro diálogo sólo puede proporcionar un comando, así que sólo debemos responder a un tipo de mensaje WM_COMMAND y al mensaje WM_INITDIALOG.
Según hemos explicado un poco más arriba, del mensaje WM_INITDIALOG debemos retornar con un valor distinto de cero si no llamamos a SetFocus, como es nuestro caso.
Este mensaje lo usaremos para inicializar nuestro diálogo antes de que sea visible para el usuario, siempre que haya algo que inicializar, claro.
Cuando procesemos el mensaje WM_COMMAND, que será siempre el que procede del único botón del diálogo, cerraremos el diálogo llamando a la función EndDialog y retornaremos con un valor distinto de cero.
En cualquier otro caso retornamos con FALSE, ya que no estaremos procesando el mensaje.
Nuestra función queda así:

```
BOOL CALLBACK DlgProc(HWND hDlg, UINT msg, WPARAM
wParam, LPARAM lParam)
{
    switch (msg)                    /* manipulador del
mensaje */
    {
        case WM_INITDIALOG:
           return TRUE;
        case WM_COMMAND:
           EndDialog(hDlg, FALSE);
           return TRUE;
    }
    return FALSE;
}
```

Bueno, sólo nos falta saber cómo creamos un cuadro de diálogo. Para ello usaremos un comando de menú, por lo tanto, el diálogo se activará desde el procedimiento de ventana.

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT
msg, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInstance;

    switch (msg)                    /* manipulador del
mensaje */
```

```
    {
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           return 0;
           break;
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case CM_DIALOGO:
                 DialogBox(hInstance,
"DialogoPrueba", hwnd, DlgProc);
                 break;
           }
           break;
        case WM_DESTROY:
           PostQuitMessage(0);         /*  envía  un
mensaje WM_QUIT a la cola de mensajes */
           break;
        default:                          /* para los
mensajes de los que no nos ocupamos */
           return  DefWindowProc(hwnd,  msg,  wParam,
lParam);
    }
    return 0;
}
```

En este procedimiento hay varias novedades:
Primero hemos declarado una variable estática "hInstance" para tener siempre a mano un manipulador de la instancia actual.
Para inicializar este valor hacemos uso del mensaje WM_CREATE, que se envía a una ventana cuando es creada, antes de que se visualice por primera vez. Aprovechamos el hecho de que nuestro procedimiento de ventana sólo recibe una vez este mensaje y de que lo hace antes de poder recibir ningún otro mensaje o comando. En el futuro veremos que se usa para toda clase de inicializaciones.
El mensaje WM_CREATE tiene como parámetro en lParam un puntero a una estructura CREATESTRUCT que contiene información sobre la ventana. En nuestro caso sólo nos interesa el campo hInstance.
La otra novedad es la llamada a la función DialogBox, que es la que crea el cuadro de diálogo.

Nota: Bueno, en realidad DialogBox no es una función, sino una macro, pero dado su formato y el modo en que se usa, la consideraremos como una función.

Esta *función* necesita varios parámetros:

1. Un manipulador a la instancia de la aplicación, que hemos obtenido al procesar el mensaje WM_CREATE.
2. Un identificador de recurso de diálogo, este es el nombre que utilizamos para el diálogo al crear el recurso, entre comillas.
3. Un manipulador a la ventana a la que pertenece el diálogo.
4. Y la dirección del procedimiento de ventana que hará el tratamiento del diálogo.

Y ya tenemos nuestro primer ejemplo del uso de diálogos, en capítulos siguientes empezaremos a conocer más detenidamente cómo usar cada uno de los controles básicos: Edit, List Box, Scroll Bar, Static, Button, Combo Box, Group Box, Check Button y Radio Button. Le dedicaremos un capítulo a cada uno de ellos.

Pasar parámetros a un cuadro de diálogo

Tenemos otra opción a la hora de crear un diálogo. En lugar de usar la macro DialogBox, podemos usar la función DialogBoxParam, que nos permite enviar un parámetro extra al procedimiento de diálogo. Este parámetro se envía a través del parámetro lParam del procedimiento de diálogo, y puede contener un valor entero, o lo que es mucho más útil, un puntero.

Esta función tiene los mismos parámetros que DialogBox, más uno añadido. Este quinto parámetro es el que podemos usar para pasar y recibir valores desde el procedimiento de diálogo.

Por ejemplo, supongamos que queremos saber cuántas veces se ha invocado un diálogo. Para ello llevaremos la cuenta en el procedimiento de ventana, incrementando esa cuenta cada vez que recivamos un comando para mostrar el diálogo. Además, pasaremos ese valor como parámetro lParam al procedimiento de diálogo.

```
    static veces;
...
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case CM_DIALOGO:
                 DialogBox(hInstance,
"DialogoPrueba", hwnd, DlgProc);
                 break;
              case CM_DIALOGO2:
                 veces++;
                 DialogBoxParam(hInstance,
"DialogoPrueba2", hwnd, DlgProc2, veces);
```

```
                break;
            }
            break;
```

Finalmente, nuestro procedimiento de diálogo tomará ese valor y lo usará para crear el texto de un control estático. (Cómo funciona esto lo veremos en otro capítulo, de momento sirva como ejemplo).

```
BOOL CALLBACK DlgProc2(HWND hDlg, UINT msg, WPARAM
wParam, LPARAM lParam)
{
    char texto[25];

    switch (msg)                    /* manipulador del
mensaje */
    {
        case WM_INITDIALOG:
           sprintf(texto,   "Veces   invocado:   %d",
lParam);
           SetWindowText(GetDlgItem(hDlg,    TEXTO),
texto);
           return TRUE;
        case WM_COMMAND:
           EndDialog(hDlg, FALSE);
           return TRUE;
    }
    return FALSE;
}
```

Hemos usado la función estándar *sprintf* para conseguir un texto estático a partir del parámetro *lParam*. Posteriormente, usamos ese texto para modificar el control estático *TEXTO*.
Usamos la misma plantilla de diálogo para ambos ejemplos, y aprovechamos el control estático para mostrar nuestro mensaje. La función SetWindowText se usa para cambiar el título de una ventana, pero también sirve para cambiar el texto de un control estático.
Cuando usemos cuadros de diálogo para pedir datos al usuario veremos que este modo de crearlos nos facilita en intercambio de datos entre la aplicación y los

procedimientos de diálogo. De otro modo tendríamos que acudir a variables globales.

Capítulo 7 Control básico Edit

Tal como hemos definido nuestro diálogo en el capítulo 6, no tiene mucha utilidad. Los diálogos se usan para intercambiar información entre la aplicación y el usuario, en ambas direcciones. El ejemplo 4 sólo lo hace en una de ellas.
En el capítulo anterior hemos usado dos controles (un texto estático y un botón), aunque sin saber exactamente cómo funcionan. En este capítulo veremos el uso del control de edición.
Un control edit es una ventana de control rectangular que permite al usuario introducir y editar texto desde el teclado.
Cuando está seleccionado muestra el texto que contiene y un cursor intermitente que indica el punto de inserción de texto. Para seleccionarlo el usuario puede hacer un click con el ratón en su interior o usar la tecla [TAB]. El usuario podrá entonces introducir texto, cambiar el punto de inserción, o seleccionar texto para ser borrado o movido usando el teclado o el ratón.
Un control de este tipo puede enviar mensajes a su ventana padre mediante WM_COMMAND, y la ventana padre puede enviar mensajes a un control edit en un cuadro de diálogo llamando a la función SendDlgItemMessage. Veremos algunos de estos mensajes en este capítulo, y el resto el capítulos más avanzados.

Fichero de recursos

Empezaremos definiendo el control edit en el fichero de recursos, y lo añadiremos a nuestro dialogo de prueba.

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END
```

```
DialogoPrueba DIALOG 0, 0, 118, 48
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
 CONTROL "Texto:", -1, "STATIC",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 28, 8
 CONTROL "", ID_TEXTO, "EDIT",
    ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
    36, 9, 76, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    56, 26, 50, 14
END
```

Hemos hecho algunas modificaciones más. Para empezar, el control static se ha convertido en una etiqueta para el control edit, que indica al usuario qué tipo de información debe suministrar.
Hemos añadido el control edit a continuación del control static. Veremos que el orden en que aparecen los controles dentro del cuadro de diálogo es muy importante, al menos en aquellos controles que tengan el estilo WS_TABSTOP, ya que ese orden será el mismo en que se activen los controles cuando usemos la tecla TAB. Para más detalles acerca de los controles edit ver controles edit.
Pero ahora veamos cómo hemos definido nuestro control edit:

```
CONTROL "", ID_TEXTO, "EDIT",
   ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
   36, 9, 76, 12
```

- CONTROL es una palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, introducimos el texto que se mostrará en el interior del control, en este caso, ninguno.
- id es el identificador del control. Los controles edit necesitan un identificador para que la aplicación pueda acceder a ellos. Usaremos un identificador definido en IDS.h.

- class es la clase de control, en nuestro caso "EDIT".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo edit y varios de ventana:
  - ES_LEFT: indica que el texto en el interior del control se alineará a la izquierda.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_BORDER: se crea un control que tiene de borde una línea fina.
  - WS_TABSTOP: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo WS_TABSTOP.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

El procedimiento de diálogo y los controles edit

Para manejar el control edit desde nuestro procedimiento de diálogo tendremos que hacer algunas modificaciones.
Para empezar, los controles edit también pueden generar mensajes WM_COMMAND, de modo que debemos diferenciar el control que originó dicho mensaje y tratarlo de diferente modo según el caso.

```
        case WM_COMMAND:
           if(LOWORD(wParam)        ==        IDOK)
EndDialog(hDlg, FALSE);
           return TRUE;
```

En nuestro caso sigue siendo sencillo: sólo cerraremos el diálogo si el mensaje WM_COMMAND proviene del botón "Aceptar".
La otra modificación afecta al mensaje WM_INITDIALOG.

```
        case WM_INITDIALOG:
           SetFocus(GetDlgItem(hDlg, ID_TEXTO));
           return FALSE;
```

De nuevo es una modificación sencilla, tan sólo haremos que el foco del teclado se coloque en el control edit, de modo que el usuario pueda empezar a escribir directamente, tan pronto como el diálogo haya aparecido en pantalla.

Para hacer eso usaremos la función SetFocus. Pero esta función requiere como parámetro el manipulador de ventana del control que debe recibir el foco, este manipulador lo conseguimos con la función GetDlgItem, que a su vez necesita como parámetros un manipulador del diálogo y el identificador del control.

Variables a editar en los cuadros de diálogo

Quizás has notado que a nuestro programa le falta algo.
Efectivamente, podemos introducir y modificar texto en el cuadro de diálogo, pero no podemos asignar valores iniciales al control de edición ni tampoco podemos hacer que la aplicación tenga acceso al texto introducido por el usuario.
Lo primero que tenemos que tener es algún tipo de variable que puedan compartir los procedimientos de ventana de la aplicación y el del diálogo. En nuestro caso se trata sólo de una cadena, pero según se añadan más parámetros al cuadro de edición, estos datos pueden ser más complejos, así que usaremos un sistema que nos valdrá en todos los casos.
Se trata de crear una estructura con todos los datos que queremos que el procedimiento de diálogo comparta con el procedimiento de ventana:

```
typedef struct stDatos {
   char Texto[80];
} DATOS;
```

Lo más sencillo es que estos datos sean globales, pero no será buena idea ya que no es buena práctica el uso de variables globales.
Tampoco parece muy buena idea declarar los datos en el procedimiento de ventana, ya que este procedimiento se usa para todas las ventanas de la misma clase, y tendríamos que definir los datos como estáticos.
Pero recordemos que tenemos un modo de pasar parámetros al cuadro de diálogo, usando la función DialogBoxParam, a través del parámetro *lParam*.
Aunque esta opción parece que nos limita a valores enteros, y sólo permite pasar valores al procedimiento de diálogo, en realidad se puede usar para pasar valores en ambos sentidos, bastará con enviar un puntero en lugar de un entero.
Para ello haremos un casting del puntero al tipo LPARAM. Dentro del procedimiento de diálogo haremos otro casting de LPARAM al puntero.
Esto nos permite declarar la variable que contiene los datos dentro del procedimiento de ventana, en este caso, de forma estática.

```
static DATOS Datos;
...
```

```
   DialogBoxParam(hInstance, "DialogoPrueba", hwnd,
DlgProc, (LPARAM)&Datos);
```

En el caso del procedimiento de diálogo:

```
    static DATOS *Datos;
...
        case WM_INITDIALOG:
           Datos = (DATOS *)lParam;
```

Daremos valores iniciales a las variables de la aplicación, dentro del procedimiento de ventana, al procesar el mensaje WM_CREATE:

```
        case WM_CREATE:
           /* Inicialización de los datos de la
aplicación */
           strcpy(Datos.Texto, "Inicial");
```

Iniciar controles edit

Ahora tenemos que hacer que se actualice el contenido del control edit al abrir el cuadro de diálogo.
El lugar adecuado para hacer esto es en el proceso del mensaje WM_INITDIALOG:

```
        case WM_INITDIALOG:
           SendDlgItemMessage(hDlg,        ID_TEXTO,
EM_LIMITTEXT, 80, 0L);
           Datos = (DATOS *)lParam;
           SetDlgItemText(hDlg,   ID_TEXTO,   Datos-
>Texto);
           SetFocus(GetDlgItem(hDlg, ID_TEXTO));
           return FALSE;
```

Hemos añadido dos llamadas a dos nuevas funciones del API. La primera es a SendDlgItemMessage, que envía un mensaje a un control. En este caso se trata de un mensaje EM_LIMITTEXT, que sirve para limitar la longitud del texto que se

puede almacenar y editar en el control edit. Es necesario que hagamos esto, ya que el texto que puede almacenar nuestra estructura de datos está limitado a 80 caracteres.
También hemos añadido una llamada a la función SetDlgItemText, que hace exactamente lo que pretendemos: cambiar el contenido del texto en el interior de un control edit.

Devolver valores a la aplicación

También queremos que cuando el usuario esté satisfecho con los datos que ha introducido, y pulse el botón de aceptar, el dato de nuestra aplicación se actualice con el texto que hay en el control edit.
Esto lo podemos hacer de varios modos. Como veremos en capítulos más avanzados, podemos responder a mensajes que provengan del control cada vez que cambia su contenido.
Pero ahora nos limitaremos a leer ese contenido cuando procesemos el comando generado al pulsar el botón de "Aceptar".

```
        case WM_COMMAND:
           if(LOWORD(wParam) == IDOK)
           {
              GetDlgItemText(hDlg,  ID_TEXTO,  Datos-
>Texto, 80);
              EndDialog(hDlg, FALSE);
           }
           return TRUE;
```

Para eso hemos añadido la llamada a la función GetDlgItemText, que es simétrica a SetDlgItemText.
Ahora puedes comprobar lo que pasa cuando abres varias veces seguidas el cuadro de diálogo modificando el texto cada vez.
Con esto parece que ya controlamos lo básico de los controles edit, pero aún hay algo más.

Añadir la opción de cancelar

Es costumbre dar al usuario la oportunidad de arrepentirse si ha modificado algo en un cuadro de diálogo y, por la razón que sea, cambia de idea.
Para eso se suele añadir un segundo botón de "Cancelar".
Empecemos por añadir dicho botón en el fichero de recursos:

```
DialogoPrueba DIALOG 0, 0, 118, 48
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
{
 CONTROL "Texto:", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 28, 8
 CONTROL "", ID_TEXTO, "EDIT",
    ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
    36, 9, 76, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_DEFPUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    8, 26, 45, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    61, 26, 45, 14
}
```

Hemos cambiado las coordenadas de los botones, para que el de "Aceptar" aparezca a la izquierda. Además, el botón de "Aceptar" lo hemos convertido en el botón por defecto, añadiendo el estilo BS_DEFPUSHBUTTON. Haciendo eso, podemos simular la pulsación del botón de aceptar pulsando la tecla de "intro".
El identificador del botón de "Cancelar" es IDCANCEL, y está definido en Windows.h.
Ahora tenemos que hacer que nuestro procedimiento de diálogo manipule el mensaje del botón de "Cancelar".

```
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case IDOK:
                 GetDlgItemText(hDlg,      ID_TEXTO,
Datos->Texto, 80);
                 EndDialog(hDlg, FALSE);
                 break;
              case IDCANCEL:
```

```
                EndDialog(hDlg, FALSE);
                break;
        }
        return TRUE;
```

Como puedes ver, sólo leemos el contenido del control edit si se ha pulsado el botón de "Aceptar".

Editar números

En muchas ocasiones necesitaremos editar valores de números enteros en nuestros diálogos.
Para eso, el API tiene previstas algunas constantes y funciones, (aunque no es así para números en coma flotante, para los que tendremos que crear nuestros propios controles).
Bien, vamos a modificar nuestro ejemplo para editar valores numéricos en lugar de cadenas de texto.

Fichero de recursos para editar enteros

Empezaremos añadiendo una constante al fichero de identificadores: "IDS.h":

```
#define ID_NUMERO 100
```

Y redefiniendo el control edit en el fichero de recursos, al que añadiremos el flag ES_NUMBER para que sólo admita caracteres numéricos:

```
DialogoPrueba DIALOG 0, 0, 118, 48
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
 CONTROL "Número:", -1, "STATIC",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 28, 8
 CONTROL "", ID_NUMERO, "EDIT",
```

```
    ES_NUMBER  |  ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |
WS_BORDER | WS_TABSTOP,
    36, 9, 76, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_DEFPUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    8, 26, 45, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    61, 26, 45, 14
END
```

Variables a editar en los cuadros de diálogo

Ahora modificaremos la estructura de los datos para que el dato a editar sea de tipo numérico:

```
typedef struct stDatos {
   int Numero;
} DATOS;
```

Al igual que antes, daremos valores iniciales a las variables del diálogo al procesar el mensaje WM_CREATE.

```
        case WM_CREATE:
           /*  Inicialización  de  los  datos  de  la
aplicación */
           Datos.Numero = 123;
```

Por supuesto, pasaremos un puntero a esta estructura a la función DialogBoxParam, haciendo uso el parámetro *lParam*:

```
static DATOS Datos;
...
```

```
   DialogBoxParam(hInstance, "DialogoPrueba", hwnd,
DlgProc, (LPARAM)&Datos);
```

Iniciar controles edit de enteros

Ahora tenemos que hacer que se actualice el contenido del control edit al abrir el cuadro de diálogo.
El lugar adecuado para hacer esto es en el proceso del mensaje WM_INITDIALOG:

```
    static DATOS *datos;
...
        case WM_INITDIALOG:
           datos = (DATOS *)lParam;
           SetDlgItemInt(hDlg,            ID_NUMERO,
(UINT)datos->Numero, FALSE);
           SetFocus(GetDlgItem(hDlg, ID_NUMERO));
           return FALSE;
```

En este caso no es necesario limitar el texto que podemos editar en el control, ya que, como veremos, las propias funciones del API se encargan de capturar y convertir el contenido del control en un número, de modo que no tenemos que preocuparnos de que no quepa en nuestra variable.
También hemos modificado la función a la que llamamos para modificar el contenido del control, ahora usaremos SetDlgItemInt, que cambia el contenido de un control edit con un valor numérico.

Devolver valores a la aplicación

Por último leeremos el contenido cuando procesemos el comando generado al pulsar el botón de "Aceptar".

```
    BOOL NumeroOk;
    int numero;
...
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case IDOK:
```

```
                numero     =     GetDlgItemInt(hDlg,
ID_NUMERO, &NumeroOk, FALSE);
                if(NumeroOk) {
                   datos->Numero = numero;
                   EndDialog(hDlg, FALSE);
                }
                else
                   MessageBox(hDlg,    "Número    no
válido", "Error",
                      MB_ICONEXCLAMATION | MB_OK);
                break;
```

Para eso hemos añadido la llamada a la función GetDlgItemInt, que es simétrica a SetDlgItemInt. El proceso difiere del usado para capturar cadenas, ya que en este caso la función nos devuelve el valor numérico del contenido del control edit.
También devuelve un parámetro que indica si ha habido algún error durante la conversión. Si el valor de ese parámetro es TRUE, significa que la conversión se realizó sin problemas, si es FALSE, es que ha habido un error. Si nuestro programa detecta un error visualizará un mensaje de error y no permitirá abandonar el cuadro de diálogo.
Pero si ha habido un error, el valor de retorno de GetDlgItemInt será cero. Esto nos causa un problema. Si leemos el valor directamente en datos->Numero y el usuario introduce un valor no válido, y después pulsa "Cancelar", el valor devuelto no será el original, sino 0. Para evitar eso hemos usado una variable local, y el valor de datos->Numero sólo se actualiza antes de salir con "Aceptar" y con un valor válido.
Por último, hemos usado el flag BM_ICONEXCLAMATION en el MessageBox, que añade un icono al cuadro de mensaje y el sonido predeterminado para alertar al usuario.

Capítulo 8 Control básico ListBox

Los controles edit son muy útiles cuando la información a introducir por el usuario es imprevisible o existen muchas opciones. Pero cuando el número de opciones no es muy grande y son todas conocidas, es preferible usar un control ListBox.
Ese es el siguiente control básico que veremos. Un ListBox consiste en una ventana rectangular con una lista de cadenas entre las cuales el usuario puede escoger una o varias.
El usuario puede seleccionar una cadena apuntándola y haciendo clic con el botón del ratón. Cuando una cadena se selecciona, se resalta y se envía un mensaje de notificación a la ventana padre. También se puede usar una barra de scroll con los listbox para desplazar listas muy largas o demasiado anchas para la ventana.

Ficheros de recursos

Empezaremos definiendo el control listbox en el fichero de recursos, y lo añadiremos a nuestro diálogo de prueba:

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 118, 135
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
 CONTROL "Lista:", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 28, 8
 CONTROL "", ID_LISTA, "listbox",
```

```
    LBS_STANDARD   |   WS_CHILD   |   WS_VISIBLE   |
WS_TABSTOP,
    9, 19, 104, 99
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_DEFPUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    8, 116, 45, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    61, 116, 45, 14
END
```

Hemos añadido el control listbox a continuación del control static. Para más detalles acerca de los controles listbox ver control listbox.
Ahora veamos cómo hemos definido nuestro control listbox:

```
CONTROL "", ID_LISTA, "listbox",
   LBS_STANDARD   |   WS_CHILD   |   WS_VISIBLE   |
WS_TABSTOP,
   9, 19, 104, 99
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los listbox no tiene ninguna función. Lo dejaremos como cadena vacía.
- id es el identificador del control. Los controles listbox necesitan un identificador para que la aplicación pueda acceder a ellos. Usaremos un identificador definido en IDS.h.
- class es la clase de control, en nuestro caso "LISTBOX".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo listbox y varios de ventana:
- LBS_STANDARD: ordena alfabéticamente las cadenas en el listbox. La ventana padre recibe in mensaje de entrada cada vez que el usuario hacer click o doble click sobre una cadena. El list box tiene bordes en todos sus lados.
- WS_CHILD: crea el control como una ventana hija.
- WS_VISIBLE: crea una ventana inicialmente visible.
- WS_TABSTOP: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo WS_TABSTOP.

- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles listbox

Para este ejemplo también usaremos variables estáticas en el procedimiento de ventana para almacenar el valor de la cadena del listbox actualmente seleccionada.

```
// Datos de la aplicación
typedef struct stDatos {
   char Item[80];
} DATOS;
```

Daremos valores iniciales a las variables, al procesar el mensaje WM_CREATE del procedimiento de ventana:

```
    static DATOS Datos;
...
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           /*  Inicialización  de  los  datos  de  la
aplicación */
           strcpy(Datos.Item, "Cadena nº 3");
           return 0;
```

Y pasaremos un puntero a la estructura con los datos como parámetro *lParam* de la función DialogBoxParam.

```
              DialogBoxParam(hInstance,
"DialogoPrueba", hwnd,
                 DlgProc, (LPARAM)&Datos);
```

La característica más importante de los listbox es que contienen listas de cadenas. Así que es imprescindible iniciar este tipo de controles, introduciendo las cadenas antes de que se muestre el diálogo. Eso se hace durante el proceso del mensaje

WM_INITDIALOG dentro del procedimiento de diálogo. En este mismo mensaje obtenemos el puntero a la estructura de los datos que recibimos en el parámetro *lParam*.

```
    static DATOS *Datos;
...
        case WM_INITDIALOG:
           Datos = (DATOS *)lParam;
           // Añadir cadenas. Mensaje: LB_ADDSTRING
           SendDlgItemMessage(hDlg,         ID_LISTA,
LB_ADDSTRING, 0, (LPARAM)"Cadena nº 1");
           SendDlgItemMessage(hDlg,         ID_LISTA,
LB_ADDSTRING, 0, (LPARAM)"Cadena nº 4");
           SendDlgItemMessage(hDlg,         ID_LISTA,
LB_ADDSTRING, 0, (LPARAM)"Cadena nº 3");
           SendDlgItemMessage(hDlg,         ID_LISTA,
LB_ADDSTRING, 0, (LPARAM)"Cadena nº 2");
           SendDlgItemMessage(hDlg,         ID_LISTA,
LB_SELECTSTRING, (UINT)-1, (LPARAM)Datos->Item);
           SetFocus(GetDlgItem(hDlg, ID_LISTA));
           return FALSE;
```

Para añadir cadenas a un listbox se usa el mensaje LB_ADDSTRING mediante la función SendDlgItemMessage, que envía un mensaje a un control.
También podemos preseleccionar alguna de las cadenas del listbox, aunque esto no es muy frecuente ya que se suele dejar al usuario que seleccione una opción sin sugerirle nada. Para seleccionar una de las cadenas también se usa un mensaje: LB_SELECTSTRING. Usaremos el valor -1 en wParam para indicar que se busque en todo el listbox.

Devolver valores a la aplicación

También queremos que cuando el usuario está satisfecho con los datos que ha introducido, y pulse el botón de aceptar, el dato de nuestra aplicación se actualice con el texto del ítem seleccionado.
De nuevo recurriremos a mensajes para pedirle al listbox el valor de la cadena actualmente seleccionada. En este caso se trata dos mensajes combinados, uno es LB_GETCURSEL, que se usa para averiguar el índice de la cadena actualmente seleccionada. El otro es LB_GETTEXT, que devuelve la cadena del índice que le indiquemos.

Cuando trabajemos con memoria dinámica y con ítems de longitud variable, será interesante saber la longitud de la cadena antes de leerla desde el listbox. Para eso podemos usar el mensaje LB_GETTEXTLEN.
Haremos esa lectura al procesar el comando IDOK, que se genera al pulsar el botón "Aceptar".

```
    UINT indice;
...
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case IDOK:
                 indice  =  SendDlgItemMessage(hDlg,
ID_LISTA, LB_GETCURSEL, 0, 0);
                 SendDlgItemMessage(hDlg,  ID_LISTA,
LB_GETTEXT, indice, (LPARAM)Datos->Item);
                 EndDialog(hDlg, FALSE);
                 break;
              case IDCANCEL:
                 EndDialog(hDlg, FALSE);
                 break;
           }
           return TRUE;
```

Capítulo 9 Control básico Button

Los controles button simulan el comportamiento de un pulsador o un interruptor. Pero sólo cuando se comportan como un pulsador los llamaremos botones, cuando emulen interruptores nos referiremos a ellos como checkbox o radiobutton.
Los botones se usan para que el usuario pueda ejecutar ciertas acciones o para dar órdenes a una aplicación. En muchos aspectos, funcionan igual que los menús, y de hecho, ambos generan mensajes de tipo WM_COMMAND.
Se componen normalmente de una pequeña área rectangular con un texto en su interior que identifica la acción que tienen asociada.
En realidad ya hemos usado controles button en todos los ejemplos anteriores, pero los explicaremos ahora con algo más de detalle.

Ficheros de recursos

Empezaremos definiendo el control button en el fichero de recursos, y lo añadiremos a nuestro dialogo de prueba:

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 130, 70
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
CAPTION "Diálogo de prueba"
FONT 8, "Helv"
BEGIN
 CONTROL "Botones:", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 9, 28, 8
 CONTROL "Nuestro botón", ID_BOTON, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    9, 19, 104, 25
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_DEFPUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    8, 50, 45, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    61, 50, 45, 14
END
```

Hemos añadido un nuevo control button a continuación del control static. Para más detalles acerca de los controles button ver controles button.
Ahora veamos cómo hemos definido nuestro control button, y también los otros dos que hemos usado hasta ahora.:

```
 CONTROL "Nuestro botón", ID_BOTON, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    9, 19, 104, 25
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_DEFPUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    8, 50, 45, 14
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los button se trata del texto que aparecerá en su interior.
- id es el identificador del control. Los controles button necesitan un identificador para que la aplicación pueda acceder a ellos y para usarlos como parámetro en los mensajes WM_COMMAND. Usaremos un identificador definido en IDS.h.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo button y varios de ventana:
  - BS_PUSHBUTTON: Crea un botón corriente que envía un mensaje WM_COMMAND a su ventana padre cuando el usuario pulsa el botón.
  - BS_DEFPUSHBUTTON: Crea un botón normal que se comporta como uno del estilo BS_PUSHBUTTON, pero también tiene un borde negro y grueso. Si el botón está en un cuadro de diálogo, el usuario puede pulsar este botón usando la tecla ENTER, aún cuando el botón no tenga el foco de entrada. Este estilo es corriente para permitir al usuario seleccionar rápidamente la opción más frecuente, la opción por defecto. Lo usaremos frecuentemente con el botón "Aceptar".
  - BS_CENTER: Centra el texto horizontalmente en el área del botón.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_TABSTOP: define un control que puede recibir el foco del teclado cuando el usuario pulsa la tecla TAB. Presionando la tecla TAB, el usuario mueve el foco del teclado al siguiente control con el estilo WS_TABSTOP.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles button

Los controles button no se usan para editar o seleccionar información, sólo para que el usuario pueda dar órdenes o indicaciones a la aplicación, así que no requieren inicialización. Lo más que haremos en algunos casos es situar el foco en uno de ellos.

Eso se hace durante el proceso del mensaje WM_INITDIALOG dentro del procedimiento de diálogo.

```
        case WM_INITDIALOG:
           SetFocus(GetDlgItem(hDlg, ID_BOTON));
           return FALSE;
```

Tratamiento de acciones de los controles button

Nuestro cuadro de diálogo tiene tres botones. Los de "Aceptar" y "Cancelar" tienen una misión clara: validar o ignorar los datos y cerrar el cuadro de diálogo. En el caso de nuestro botón, queremos que se realice otra operación diferente, por ejemplo, mostrar un mensaje.

```
        case WM_COMMAND:
           switch(LOWORD(wParam)) {
              case ID_BOTON:
                 MessageBox(hDlg, "Se pulsó 'Nuestro
botón'", "Acción", MB_ICONINFORMATION|MB_OK);
                 break;
              case IDOK:
                 EndDialog(hDlg, FALSE);
                 break;
              case IDCANCEL:
                 EndDialog(hDlg, FALSE);
                 break;
           }
           return TRUE;
```

Capítulo 10 Control básico Static

Los static son el tipo de control menos interactivo, normalmente se usan como información o decoración. Pero son muy importantes. Windows es un entorno gráfico, y la apariencia forma una parte muy importante de él.

Existen varios tipos de controles static, o mejor dicho, varios estilos de controles static.

Dependiendo del estilo que elijamos para cada control static, su aspecto será radicalmente distinto, desde una simple línea o cuadro hasta un bitmap, un icono o un texto.

Cuando hablemos de controles static de tipo texto, frecuentemente nos referiremos a ellos como etiquetas. Las etiquetas pueden tener también una función añadida, como veremos más adelante: nos servirán para acceder a otros controles usando el teclado.

En realidad ya hemos usado controles static del tipo etiqueta cuando vimos los controles edit, lisbox y button, pero de nuevo los explicaremos ahora con más detalle.

Ficheros de recursos

Empezaremos definiendo varios controles static en el fichero de recursos, y los añadiremos a nuestro dialogo de prueba, para obtener un muestrario:

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 240, 120
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION
// STYLE DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP
| WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Prueba de static"
FONT 8, "Helv"
BEGIN
 CONTROL "Frame1", -1, "STATIC",
    SS_WHITEFRAME | WS_CHILD | WS_VISIBLE,
    8, 5, 52, 34
 CONTROL "Frame2", -1, "STATIC",
    SS_GRAYFRAME | WS_CHILD | WS_VISIBLE,
    12, 9, 52, 34
 CONTROL "Frame3", -1, "STATIC",
    SS_BLACKFRAME | WS_CHILD | WS_VISIBLE,
    16, 13, 52, 34
 CONTROL "Rect1", -1, "STATIC",
    SS_BLACKRECT | WS_CHILD | WS_VISIBLE,
    72, 22, 48, 34
 CONTROL "Rect2", -1, "STATIC",
    SS_GRAYRECT | WS_CHILD | WS_VISIBLE,
```

```
    12, 60, 52, 34
 CONTROL "Rect3", -1, "STATIC",
    SS_WHITERECT | WS_CHILD | WS_VISIBLE,
    72, 60, 48, 34
 CONTROL "Bitmap1", -1, "STATIC",
    SS_BITMAP | WS_CHILD | WS_VISIBLE,
    128, 22, 18, 15
 CONTROL "Icono", -1, "STATIC",
    SS_ICON | WS_CHILD | WS_VISIBLE,
    188, 47, 20, 20
 CONTROL "Edit &1:", -1, "STATIC",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    128, 73, 40, 9
 CONTROL "", ID_EDIT1, "EDIT",
    ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
    180, 73, 20, 12
 CONTROL "Edit &2:", -1, "STATIC",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    128, 95, 28, 8
 CONTROL "", ID_EDIT2, "EDIT",
    ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
    180, 95, 20, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    186, 6, 50, 14
END
```

Hemos añadido diez nuevos controles static. Para más detalles acerca de los controles static ver controles static.

Ahora veamos cómo hemos definido nuestros controles static:

```
CONTROL  "Frame1",  -1,  "static",  SS_WHITEFRAME  |
WS_CHILD | WS_VISIBLE, 8, 5, 52, 34
CONTROL   "Bitmap1",   -1,   "static",   SS_BITMAP   |
WS_CHILD | WS_VISIBLE, 128, 22, 18, 15
CONTROL "Icono", -1, "static", SS_ICON | WS_CHILD |
WS_VISIBLE, 188, 47, 20, 20
```

```
CONTROL "Edit &1:", -1, "static", SS_LEFT | WS_CHILD
| WS_VISIBLE, 128, 73, 40, 9
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los static tiene sentido para las etiquetas, los bitmaps y los iconos. En estos dos últimos casos indicará el nombre del recurso a insertar. En el resto de los casos se incluye como información. Comentaremos algo más sobre los textos de la etiquetas más abajo.
- id es el identificador del control. Los controles static no suelen necesitar un identificador, ya que no suelen tener un comportamiento interactivo. De modo que todos los identificadores de controles static serán -1.
- class es la clase de control, en nuestro caso "STATIC".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo static y varios de ventana:
  - SS_WHITEFRAME, SS_GRAYFRAME, SS_BLACKFRAME: Crea un rectángulo vacío o un marco de color blanco, gris o negro, respectivamente. SS_WHITEREC, SS_GRAYREC, SS_BLACKREC: Crea un rectángulo relleno de color blanco, gris o negro, respectivamente. SS_BITMAP mostrará el mapa de bits indicado en el campo text. SS_ICON mostrará el icono indicado en el campo text. SS_LEFT, SS_RIGHT, SS_CENTER: indican que es una etiqueta y ajustará el texto a la izquierda, la derecha o el centro, respectivamente.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

En el caso de las etiquetas, cuando se incluye el carácter '&', el siguiente carácter de la cadena aparecerá subrayado, indicando que puede ser usado como acelerador, (pulsando la tecla [ALT] más el carácter subrayado), para acceder al control más cercano, normalmente a su derecha o debajo. Pero, cuidado, en realidad el acelerador situará el foco en el control definido exactamente a continuación del control static en el fichero de recursos, y no al más cercano físicamente en pantalla.

En el ejemplo, verifica lo que sucede al pulsar la tecla ALT más '1' ó '2'. Verás que el foco del teclado se desplaza a los cuadros de edición 1 y 2.

Iniciar controles static

Los controles static normalmente no necesitan iniciase, por algo son estáticos. Sin embargo, a veces necesitaremos modificar el texto de alguna etiqueta, esto puede ser útil para mostrar alguna información en un cuadro de diálogo, por ejemplo. Para eso podemos usar la misma función que en los controles edit: SetDlgItemText. En este caso, necesitaremos usar un identificador válido para el control estático.

Tratamiento de acciones de los controles static

Los controles static tampoco responderán, normalmente, a acciones del usuario, ni tampoco generarán mensajes. En el caso de las etiquetas, el comportamiento de los aceleradores es automático y no requerirá ninguna acción del programa.

Capítulo 11 Control básico ComboBox

Los ComboBoxes son una combinación de un control Edit y un Lisbox. Son los controles que suelen recordar las entradas que hemos introducido antes, para que podamos seleccionarlas sin tener que escribirlas de nuevo, en ese sentido funcionan igual que un Listbox, pero también permiten introducir nuevas entradas.
Hay modalidades de ComboBox en las que el control Edit está inhibido, y no permite introducir nuevos valores. En esos casos, el control se comportará de un modo muy parecido al que lo hace un Listbox, pero, como veremos más adelante, tienen ciertas ventajas.
Exiten tres tipos de ComboBoxes:

- Simple: es la forma que muestra siempre el control Edit y el ListBox, aunque ésta esté vacía.
- DropDown: despliegue hacia abajo. Se muestra un pequeño icono a la derecha del control Edit. Si el usuario lo pulsa con el ratón, se desplegará el ListBox, mientras no se pulse, la lista permanecerá oculta.
- DropDownList: lo mismo, pero el control Edit se sustituye por un control Static.

Ficheros de recursos

Para nuestro ejemplo incluiremos un control ComboBox de cada tipo, así veremos las peculiaridades de cada uno:

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 205, 78
```

```
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION | WS_SYSMENU
CAPTION "Combo boxes"
FONT 8, "Helv"
BEGIN
 CONTROL "&Simple", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    8, 2, 60, 8
 CONTROL "ComboBox1", ID_COMBOBOX1, "COMBOBOX",
    CBS_SORT | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
    8, 13, 60, 43
 CONTROL "&Dropdown", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    73, 2, 60, 8
 CONTROL "ComboBox2", ID_COMBOBOX2, "COMBOBOX",
    CBS_DROPDOWN | CBS_SORT | WS_CHILD | WS_VISIBLE
| WS_TABSTOP,
    72, 13, 60, 103
 CONTROL "Dropdown &List", -1, "static",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    138, 2, 60, 8
 CONTROL "ComboBox3", ID_COMBOBOX3, "COMBOBOX",
    CBS_DROPDOWNLIST  |  CBS_SORT  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    136, 13, 60, 103
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    28, 60, 50, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    116, 60, 50, 14
END
```

Hemos añadido los nuevos controles ComboBox. Para más detalles acerca de estos controles ver controles combobox.

Ahora veremos más detalles sobre los estilos de los controles ComboBox:

```
 CONTROL "ComboBox1", ID_COMBOBOX1, "COMBOBOX",
    CBS_SORT | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
```

```
    8, 13, 60, 43
 CONTROL "ComboBox2", ID_COMBOBOX2, "COMBOBOX",
    CBS_DROPDOWN | CBS_SORT | WS_CHILD | WS_VISIBLE
| WS_TABSTOP,
    72, 13, 60, 103
 CONTROL "ComboBox3", ID_COMBOBOX3, "COMBOBOX",
    CBS_DROPDOWNLIST   |   CBS_SORT   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    136, 13, 60, 103
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los combobox sólo sirve como información, y no se usa.
- id es el identificador del control. El identificador será necesario para inicializar y leer los contenidos y selecciones del combobox.
- class es la clase de control, en nuestro caso "COMBOBOX".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo combobox y varios de ventana:
  - CBS_SORT: Indica que los valores en la lista se apareceran por orden alfabético. CBS_DROPDOWN crea un ComboBox del tipo DropDown. CBS_DROPDOWNLIST crea un ComboBox del tipo DropDownList.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_TABSTOP: para que cuando el foco cambie de control al pulsar TAB, pase por este control.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles ComboBox

Iniciar los controles ComboBox es análogo a iniciar los controles ListBox. En general, necesitaremos introducir una lista de valores en el listbox, para que el usuario pueda usarla.

El lugar adecuado para hacerlo también es al procesar el mensaje WM_INITDIALOG de nuestro cuadro de diálogo, y el mensaje para añadir cadenas es CB_ADDSTRING. Hay que recordar también que para enviar mensajes a un control se usa la función SendDlgItemMessage.

Para hacer más fácil la inicialización de las listas, usaremos los mismos valores en las tres. Para ello definiremos, dentro de nuestra estructura de datos para compartir con el cuadro de diálogo, un array de cadenas con los valores necesario para inicializar los ComboBox:

También usaremos la misma estructura para almacenar los valores iniciales y de la última selección de los tres comboboxes:

```
/* Datos de la aplicación */
typedef struct stDatos {
   char Lista[6][80]; // Valores de los comboboxes
   char Item[3][80]; // Opciones elegidas
} DATOS;
```

Y asignaremos valores iniciales a las selecciones y a la lista de opciones al procesar el mensaje WM_CREATE:

```
    static DATOS Datos;
    int i;
...
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           strcpy(Datos.Item[0], "a");
           strcpy(Datos.Item[1], "c");
           strcpy(Datos.Item[2], "e");
           for(i = 0; i < 6; i++)
              sprintf(Datos.Lista[i],   "%c)   Opción
%c", 'a'+i, 'A'+i);
           return 0;
```

La parte de inicialización de los comboboxes se hace al procesar el mensaje WM_INITDIALOG:

```
    int i;
    static DATOS *Datos;
...
        case WM_INITDIALOG:
           Datos = (DATOS*)lParam;
           // Añadir cadenas. Mensaje: LB_ADDSTRING
           for(i = 0; i < 6; i++) {
              SendDlgItemMessage(hDlg, ID_COMBOBOX1,
                 CB_ADDSTRING,   0,   (LPARAM)Datos-
>Lista[i]);
              SendDlgItemMessage(hDlg, ID_COMBOBOX2,
                 CB_ADDSTRING,   0,   (LPARAM)Datos-
>Lista[i]);
              SendDlgItemMessage(hDlg, ID_COMBOBOX3,
```

```
                CB_ADDSTRING,   0,   (LPARAM)Datos-
>Lista[i]);
            }
            SendDlgItemMessage(hDlg,    ID_COMBOBOX1,
CB_SELECTSTRING,
               (WPARAM)-1, (LPARAM)Datos->Item[0]);
            SendDlgItemMessage(hDlg,    ID_COMBOBOX2,
CB_SELECTSTRING,
               (WPARAM)-1, (LPARAM)Datos->Item[1]);
            SendDlgItemMessage(hDlg,    ID_COMBOBOX3,
CB_SELECTSTRING,
               (WPARAM)-1, (LPARAM)Datos->Item[2]);
            SetFocus(GetDlgItem(hDlg, ID_COMBOBOX1));
            return FALSE;
```

También podemos preseleccionar alguna de las cadenas de los comboboxes. Para seleccionar una de las cadenas se usa un mensaje: CB_SELECTSTRING. Usaremos el valor -1 en wParam para indicar que se busque en toda la lista.

Devolver valores a la aplicación

Por supuesto, querremos que cuando el usuario esté satisfecho con los datos que ha introducido, y pulse el botón de aceptar, el dato de nuestra aplicación se actualice con el texto del ítem seleccionado.

De nuevo podemos recurrir a mensajes para pedirle al combobox el valor de la cadena actualmente seleccionada. En este caso se trata dos mensajes combinados, análogos a los usados en los Listbox. Uno es CB_GETCURSEL, que se usa para averiguar el índice de la cadena actualmente seleccionada.

El otro es CB_GETLBTEXT, que devuelve la cadena del índice que le indiquemos.

Cuando trabajemos con memoria dinámica y con ítems de longitud variable, será interesante saber la longitud de la cadena antes de leerla desde el listbox. Para eso podemos usar el mensaje CB_GETLBTEXTLEN.

Pero esto es válido para comboboxes del tipo *DropDownList*, que se comportan como un Listbox, sin embargo en las otras modalidades de controles ComboBox, el ítem seleccionado no tiene por qué ser igual que el texto que contiene el control Edit.

Para capturar el contenido del control Edit asociado a un ComboBox se puede usar la función GetDlgItemText. Y también el mensaje WM_GETTEXT y WM_GETTEXTLENGTH.

Como tenemos tres tipos de ComboBox, usaremos un método diferente con cada uno de ellos. Veamos cómo podría quedar el tratamiento del mensaje WM_COMMAND:

```
        case WM_COMMAND:
```

```
            switch(LOWORD(wParam)) {
               case IDOK:
                  // En el ComboList Simple usaremos:
                  GetDlgItemText(hDlg,   ID_COMBOBOX1,
Datos->Item[0], 80);
                  //   En   el   ComboList   DropDown
usaremos:
                  SendDlgItemMessage(hDlg,
ID_COMBOBOX2, WM_GETTEXT,
                     80, (LPARAM)Datos->Item[1]);
                  //  En  el  ComboList  DropDownList
usaremos:
                  indice  =  SendDlgItemMessage(hDlg,
ID_COMBOBOX3,
                     CB_GETCURSEL, 0, 0);
                  SendDlgItemMessage(hDlg,
ID_COMBOBOX3,
                     CB_GETLBTEXT,            indice,
(LPARAM)Datos->Item[2]);
                  wsprintf(resultado, "%s\n%s\n%s",
                     Datos->Item[0],  Datos->Item[1],
Datos->Item[2]);
                  MessageBox(hDlg,         resultado,
"Leido", MB_OK);
                  EndDialog(hDlg, FALSE);
                  return TRUE;
               case IDCANCEL:
                  EndDialog(hDlg, FALSE);
                  return FALSE;
```

Pero ahora surge un problema. Si en un combobox introducimos una cadena que no está en nuestra lista, y posteriormente volvemos a entrar en el cuadro de diálogo, no podremos editar el valor inicial. Es más, ni siquiera nos será mostrado.
Para evitar eso deberíamos añadir cada nuevo valor introducido a la lista. Nuestro ejemplo es un poco limitado, ya que no tiene previsto que la lista pueda crecer, y desde luego, no guardará los valores de la lista cuando el programa termine, de modo que estén disponibles en sucesivas ejecuciones. Pero de momento nos conformaremos con ciertas modificaciones mínimas que ilustren cómo solventar este error.
Para empezar, reservaremos espacio suficiente para almacenar algunos valores extra, y modificaremos la estructura de datos:

```
#define MAX_CADENAS 100
...

/* Datos de la aplicación */
typedef struct stDatos {
   int nCadenas;
   char Lista[MAX_CADENAS][80];
   char Item[3][80];
} DATOS;
```

También deberemos inicializar el valor de nCadenas, al procesar el mensaje WM_CREATE:

```
            Datos.nCadenas = 6;
```

Modificaremos la rutina para inicializar los ComboBoxes:

```
          // Añadir cadenas. Mensaje: LB_ADDSTRING
          for(i = 0; i < Datos->nCadenas; i++) {
             SendDlgItemMessage(hDlg, ID_COMBOBOX1,
                CB_ADDSTRING,   0,   (LPARAM)Datos-
>Lista[i]);
             SendDlgItemMessage(hDlg, ID_COMBOBOX2,
                CB_ADDSTRING,   0,   (LPARAM)Datos-
>Lista[i]);
             SendDlgItemMessage(hDlg, ID_COMBOBOX3,
                CB_ADDSTRING,   0,   (LPARAM)Datos-
>Lista[i]);
          }
...
```

Y para leer los valores introducidos, y añadirlos a la lista si no están. Para eso usaremos el mensaje CB_FINDSTRINGEXACT, que buscará una cadena entre los valores almacenados en la lista, si se encuentra devolverá un índice, y si no, el valor CB_ERR.

Existe otro mensaje parecido, CB_FINDSTRING, pero no nos vale, porque localizará la primera cadena de la lista que comience con los mismos caracteres que la cadena que buscamos. Por ejemplo, si hay un valor en la lista "valor a", y nosotros buscamos "valor", obtendremos el índice de la cadena "valor a", que no es lo que queremos, al menos en este ejemplo.

El proceso del comando IDOK quedaría así:

```
            case IDOK:
```

```
                // En el ComboList Simple usaremos:
                GetDlgItemText(hDlg,   ID_COMBOBOX1,
Datos->Item[0], 80);
                if(SendDlgItemMessage(hDlg,
ID_COMBOBOX1, CB_FINDSTRINGEXACT,
                   (WPARAM)-1,       (LPARAM)Datos-
>Item[0]) == CB_ERR)
                   strcpy(Datos->Lista[Datos-
>nCadenas++], Datos->Item[0]);
                //   En   el   ComboList   DropDown
usaremos:
                SendDlgItemMessage(hDlg,
ID_COMBOBOX2, WM_GETTEXT,
                   80, (LPARAM)Datos->Item[1]);
                if(SendDlgItemMessage(hDlg,
ID_COMBOBOX1, CB_FINDSTRINGEXACT,
                   (WPARAM)-1,       (LPARAM)Datos-
>Item[1]) == CB_ERR &&
                   strcmp(Datos->Item[0],    Datos-
>Item[1]))
                   strcpy(Datos->Lista[Datos-
>nCadenas++], Datos->Item[1]);
                //  En  el  ComboList  DropDownList
usaremos:
                indice  =  SendDlgItemMessage(hDlg,
ID_COMBOBOX3,
                   CB_GETCURSEL, 0, 0);
                SendDlgItemMessage(hDlg,
ID_COMBOBOX3,
                   CB_GETLBTEXT,            indice,
(LPARAM)Datos->Item[2]);
                wsprintf(resultado, "%s\n%s\n%s",
                   Datos->Item[0],   Datos->Item[1],
Datos->Item[2]);
                MessageBox(hDlg,         resultado,
"Leido", MB_OK);
                EndDialog(hDlg, FALSE);
                return TRUE;
```

Capítulo 12 Control básico Scrollbar

Veremos ahora el siguiente control básico: la barra de desplazamiento o Scrollbar.
Las ventanas pueden mostrar contenidos que ocupan más espacio del que cabe en su interior, cuando eso sucede se suelen agregar unos controles en forma de barra que permiten desplazar el contenido a través del área de la ventana de modo que el usuario pueda ver las partes ocultas del documento.
Pero las barras de scroll pueden usarse para introducir otros tipos de datos en nuestras aplicaciones, en general, cualquier magnitud de la que sepamos el máximo y el mínimo, y que tenga un rango valores finito. Por ejemplo un control de volumen, de 0 a 10, o un termostato de -15º a 60º.
Las barras de desplazamiento tienen varias partes o zonas diferenciadas, cada una con su función particular. Me imagino que ya las conoces, pero las veremos desde el punto de vista de un programador.
Una barra de desplazamiento consiste en un rectángulo sombreado con un botón de flecha en cada extremo, y una caja en el interior del rectángulo (llamado normalmente thumb). La barra de desplazamiento representa la longitud o anchura completa del documento, y la caja interior la porción visible del documento dentro de la ventana. La posición de la caja cambia cada vez que el usuario desplaza el documento para ver diferentes partes de él. También se modifica el tamaño de la caja para adaptarlo a la proporción del documento que es visible. Cuanta más porción del documento resulte visible, mayor será el tamaño de la caja, y viceversa.
Hay dos modalidades de ScrollBars: horizontales y verticales.
El usuario puede desplazar el contenido de la ventana pulsando uno de los botones de flecha, pulsando en la zona sombreada no ocupada por el thumb, o desplazando el propio thumb. En el primer caso se desplazará el equivalente a una unidad (si es texto, una línea o columna). En el segundo, el contenido se desplazará en la porción equivalente al contenido de una ventana. En el tercer caso, la cantidad de documento desplazado dependerá de la distancia que se desplace el thumb.
Hay que distinguir los controles ScrollBar de las barras de desplazamiento estándar. Aparentemente son iguales, y se comportan igual, los primeros están en el área de cliente de la ventana, pero las segundas no, éstas se crean y se muestran junto con la ventana. Para añadir estas barras a tu ventana, basta con crearla con los estilos WS_HSCROLL, WS_VSCROLL o ambos. WS_HSCROLL añade una barra horizontal y WS_VSCROLL una vertical.
Un control scroll bar es una ventana de contol de la clase SCROLLBAR. Se pueden crear tantas barras de scroll como se quiera, pero el programador es el encargado de especidicar el tamaño y la posición de la barra.

Ficheros de recursos

Para nuestro ejemplo incluiremos un control ScrollBar de cada tipo, aunque en realidad son casi idénticos en comportamiento:

```
#include <windows.h>
#include "IDS.H"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 189, 106
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION | WS_SYSMENU
CAPTION "Scroll bars"
FONT 8, "Helv"
BEGIN
 CONTROL "ScrollBar1", ID_SCROLLH, "SCROLLBAR",
    SBS_HORZ | WS_CHILD | WS_VISIBLE,
    7, 3, 172, 9
 CONTROL "Scroll 1:", -1, "STATIC",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    24, 18, 32, 8
 CONTROL "Edit1", ID_EDITH, "EDIT",
    ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
    57, 15, 32, 12
 CONTROL "ScrollBar2", ID_SCROLLV, "SCROLLBAR",
    SBS_VERT | WS_CHILD | WS_VISIBLE,
    7, 15, 9, 86
 CONTROL "Scroll 2:", -1, "STATIC",
    SS_LEFT | WS_CHILD | WS_VISIBLE,
    23, 41, 32, 8
 CONTROL "Edit2", ID_EDITV, "EDIT",
    ES_LEFT  |  WS_CHILD  |  WS_VISIBLE  |  WS_BORDER  |
WS_TABSTOP,
    23, 51, 32, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
```

```
    40, 87, 50, 14
END
```

Para ver cómo funcionan las barras de scroll hemos añadido dos controles Edit, que mostrarán los valores seleccionados en cada control ScrollBar. Para más detalles acerca de estos controles ver control scrollbar.
Ahora veremos más cosas sobre los estilos de los controles ScrollBar:

```
CONTROL "ScrollBar1", ID_SCROLLH, "SCROLLBAR",
   SBS_HORZ | WS_CHILD | WS_VISIBLE,
   7, 3, 172, 9
CONTROL "ScrollBar2", ID_SCROLLV, "SCROLLBAR",
   SBS_VERT | WS_CHILD | WS_VISIBLE,
   7, 15, 9, 86
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los scrollbar sólo sirve como información, y no se usa.
- id es el identificador del control. El identificador será necesario para inicializar y leer el valor del scrollbar, así como para manipular los mensajes que produzca.
- class es la clase de control, en nuestro caso "SCROLLBAR".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo scrollbar y varios de ventana:
  - SBS_HORZ: Indica se trata de un scrollbar horizontal.
  - SBS_VERT: Indica se trata de un scrollbar vertical.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles Scrollbar

Los controles Scrollbar tienen varios tipos de parámetros que hay que iniciar. Los límites de valores mínimo y máximo, y también el valor actual.
El lugar adecuado para hacerlo también es al procesar el mensaje WM_INITDIALOG de nuestro cuadro de diálogo, y para ajustar los parámetros podemos usar mensajes o funciones. En el caso de hacerlo con mensajes hay que usar la función SendDlgItemMessage.
También usaremos una estructura para almacenar los valores iniciales y de la última selección de las dos barras de desplazamiento:

```
// Datos de la aplicación
typedef struct stDatos {
```

```
   int ValorH;
   int ValorV;
} DATOS;
```

Declararemos los datos como estáticos en el procedimiento de ventana, y asignaremos valores iniciales a los controles al procesar el mensaje WM_CREATE:

```
    static DATOS Datos;
...
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           Datos.ValorH = 10;
           Datos.ValorV = 32;
           return 0;
```

Pasaremos un puntero a nuestra estructura de datos al procedimiento de diálogo usando el parámetros lParam y la función DialogBoxParam:

```
DialogBoxParam(hInstance,   "DialogoPrueba",   hwnd,
DlgProc, (LPARAM)&Datos);
```

En el procedimiento de diálogo disponemos de un puntero estático a la estructura de datos, que inicializaremos al procesar el mensaje WM_INITDIALOG.
La parte de inicialización de los scrollbars es como sigue. Hemos inicializado el scrollbar horizontal usando las funciones y el vertical usando los mensajes:

```
    static DATOS *Datos;
...
        case WM_INITDIALOG:
           Datos = (DATOS*)lParam;
           SetScrollRange(GetDlgItem(hDlg,
ID_SCROLLH), SB_CTL,
              0, 100, TRUE);
           SetScrollPos(GetDlgItem(hDlg,
ID_SCROLLH), SB_CTL,
              Datos->ValorH, TRUE);
           SetDlgItemInt(hDlg,             ID_EDITH,
(UINT)Datos->ValorH, FALSE);
           SendDlgItemMessage(hDlg,      ID_SCROLLV,
SBM_SETRANGE,
```

```
            (WPARAM)0, (LPARAM)50);
         SendDlgItemMessage(hDlg,      ID_SCROLLV,
SBM_SETPOS,
            (WPARAM)Datos->ValorV, (LPARAM)TRUE);
         SetDlgItemInt(hDlg,            ID_EDITV,
(UINT)Datos->ValorV, FALSE);
```

Para iniciar el rango de valores del scrollbar se usar la función SetScrollRange o el mensaje SBM_SETRANGE. Para cambiar el valor seleccionado o posición se usa la función SetScrollPos o el mensaje SBM_SETPOS.

Iniciar controles scrollbar: estructura SCROLLINFO

A partir de la versión 4.0 de Windows existe otro mecanismo para inicializar los scrollbars. También tiene la doble forma de mensaje y función. Su uso se recomienda en lugar de SetScrollRange, que sólo se conserva por compatibilidad con Windows 3.x.
Se trata de la función SetScrollInfo y del mensaje SBM_SETSCROLLINFO. También es necesaria una estructura que se usará para pasar los parámetros tanto a la función como al mensaje: SCROLLINFO.
Usando esta forma, el ejemplo anterior quedaría así:

```
    static DATOS *Datos;
    SCROLLINFO sih = {
       sizeof(SCROLLINFO),
       SIF_POS | SIF_RANGE | SIF_PAGE,
       0, 104,
       5,
       0,
       0};
    SCROLLINFO siv = {
       sizeof(SCROLLINFO),
       SIF_POS | SIF_RANGE | SIF_PAGE,
       0, 54,
       5,
       0,
       0};

...
        case WM_INITDIALOG:
           Datos = (DATOS*)lParam;
```

```
            sih.nPos = Datos->ValorH;
            siv.nPos = Datos->ValorV;
            SetScrollInfo(GetDlgItem(hDlg,
ID_SCROLLH), SB_CTL, &sih, TRUE);
            SetDlgItemInt(hDlg,             ID_EDITH,
(UINT)Datos->ValorH, FALSE);
            SendDlgItemMessage(hDlg,      ID_SCROLLV,
SBM_SETSCROLLINFO,
                  (WPARAM)TRUE, (LPARAM)&siv);
            SetDlgItemInt(hDlg,             ID_EDITV,
(UINT)Datos->ValorV, FALSE);
            return FALSE;
```

El segundo campo de la estructura SCROLLINFO consiste en varios bits que indican qué parámetros de la estructura se usarán para inicializar los scrollbars. Hemos incluido la posición, el rango y el valor de la página. Sería equivalente haber puesto únicamente SIF_ALL.

El valor de la página no lo incluíamos antes, y veremos que será útil al procesar los mensajes que provienen de los controles scrollbar. Además el tamaño de la caja de desplazamiento se ajusta de modo que esté a escala en relación con el tamaño total del control scrollbar. Si hubiéramos definido una página de 50 y un rango de 0 a 100, el tamaño de la caja sería exactamente la mitad del tamaño del scrollbar.

Hay que tener en cuenta que el valor máximo que podremos establecer en un control no es siempre el que nosotros indicamos en el miembro nMax de la estructura SCROLLINFO. Este valor depende del valor de la página (nPage), y será nMax-nPage+1. Así que si queremos que nuestro control pueda devolver 100, y la página tiene un valor de 5, debemos definir nMax como 104. Este funcionamiento está diseñado para scrollbars como los que incluyen las ventanas, donde la caja indica la porción del documento que se muestra en su interior.

Procesar los mensajes procedentes de controles Scrollbar

Cada vez que el usuario realiza alguna acción en un control Scrollbar se envía un mensaje WM_HSCROLL o WM_VSCROLL, dependiendo del tipo de control, a la ventana donde está insertado. En realidad eso pasa con todos los controles, pero en el caso de los Scrollbars, es imprescindible que el programa procese algunos de esos mensajes adecuadamente.

Estos mensajes entregan distintos valores en la palabra de menor peso del parámetro wParam, según la acción del usuario sobre el control. En la palabra de mayor peso se incluye la posición actual y en lParam el manipulador de ventana del control.

De modo que nuestra rutina para manejar los mensajes de los scrollbars debe ser capaz de distinguir el control del que procede el mensaje y el tipo de acción, para actuar en consecuencia.
En nuestro caso es irrelevante la orientación de la barra de scroll, podemos distinguirlos por el identificador de ventana, de todos modos procesaremos cada uno de los dos mensajes con una rutina distinta.
Para no recargar en exceso el procedimiento de ventana del diálogo, crearemos una función para procesar los mensajes de las barras de scroll. Y la llamaremos al recibir esos mensajes:

```
    switch (msg)                    /* manipulador del
mensaje */
    {
...
        case WM_HSCROLL:
           ProcesarScrollH(hDlg, (HWND)lParam,
              (int)LOWORD(wParam),
(int)HIWORD(wParam));
           return FALSE;
        case WM_VSCROLL:
           ProcesarScrollV(hDlg, (HWND)lParam,
              (int)LOWORD(wParam),
(int)HIWORD(wParam));
           return FALSE;
...
```

Los códigos que tenemos que procesar son los siguientes:

- SB_BOTTOM desplazamiento hasta el principio de la barra, en verticales arriba y en horizontales a la izquierda.
- SB_TOP: desplazamiento hasta el final de la barra, en verticales abajo y en horizontales a la derecha.
- SB_LINERIGHT y SB_LINEDOWN: desplazamiento una línea a la derecha en horizontales o abajo en verticales.
- SB_LINELEFT y SB_LINEUP: desplazamiento un línea a la izquierda en horizontales o arriba en verticales.
- SB_PAGERIGHT y SB_PAGEDOWN: desplazamiento de una página a la derecha en horizontales y abajo en verticales.
- SB_PAGELEFT y SB_PAGEUP: desplazamiento un párrafo a la izquierda en horizontales o arriba en verticales.
- SB_THUMBPOSITION: se envía cuando el thumb está en su posición final.
- SN_THUMBTRACK: el thumb se está moviendo.
- SB_ENDSCROLL: el usuario a liberado el thumb en una nueva posición.

En nuestro caso, no haremos que la variable asociada se actualice hasta que pulsemos el botón de aceptar, pero actualizaremos la posición del thumb y el valor del control edit asociado a cada scrollbar.

Veamos por ejemplo la rutina para tratar el scroll horizontal:

```
void ProcesarScrollH(HWND hDlg, HWND Control, int
Codigo, int Posicion)
{
   int Pos = GetScrollPos(Control, SB_CTL);

   switch(Codigo) {
      case SB_BOTTOM:
         Pos = 0;
         break;
      case SB_TOP:
         Pos = 100;
         break;
      case SB_LINERIGHT:
         Pos++;
         break;
      case SB_LINELEFT:
         Pos--;
         break;
      case SB_PAGERIGHT:
         Pos += 5;
         break;
      case SB_PAGELEFT:
         Pos -= 5;
         break;
      case SB_THUMBPOSITION:
      case SB_THUMBTRACK:
         Pos = Posicion;
      case SB_ENDSCROLL:
         break;
   }
   if(Pos < 0) Pos = 0;
   if(Pos > 100) Pos = 100;
   SetDlgItemInt(hDlg, ID_EDITH, (UINT)Pos, FALSE);

   SetScrollPos(Control, SB_CTL, Pos, TRUE);
}
```

Como puede observarse, actualizamos el valor de la posición dependiendo del código recibido. Nos aseguramos de que está dentro de los márgenes permitidos y

finalmente actualizamos el contenido del control edit. La función para el scrollbar vertical es análoga, pero cambiando los identificadores de los códigos y los valores de los límites.
Hemos usado una función nueva, GetScrollPos para leer la posición actual del thumb.

Procesar mensajes de scrollbar usando SCROLLINFO

Como comentamos antes, a partir de la versión 4.0 de Windows existe otro mecanismo para inicializar los scrollbars. También tiene la doble forma de mensaje y función. Su uso se recomienda en lugar de GetScrollRange, que sólo se conserva por compatibilidad con Windows 3.x.
Usando GetScrollInfo, la función para procesar el mensaje del scrollbar horizontal quedaría así:

```
void ProcesarScrollH(HWND hDlg, HWND Control, int
Codigo, int Posicion)
{
   SCROLLINFO si = {
       sizeof(SCROLLINFO),
       SIF_ALL, 0, 0, 0, 0, 0};

   GetScrollInfo(Control, SB_CTL, &si);

   switch(Codigo) {
      case SB_BOTTOM:
         si.nPos = si.nMin;
         break;
      case SB_TOP:
         si.nPos = si.nMax;
         break;
      case SB_LINEDOWN:
         si.nPos++;
         break;
      case SB_LINEUP:
         si.nPos--;
         break;
      case SB_PAGEDOWN:
         si.nPos += si.nPage;
         break;
      case SB_PAGEUP:
         si.nPos -= si.nPage;
```

```
         break;
      case SB_THUMBPOSITION:
      case SB_THUMBTRACK:
         si.nPos = Posicion;
      case SB_ENDSCROLL:
         break;
   }
   if(si.nPos < si.nMin) si.nPos = si.nMin;
   if(si.nPos   >   si.nMax-si.nPage+1)   si.nPos   =
si.nMax-si.nPage+1;

   SetScrollInfo(Control, SB_CTL, &si, true);
   SetDlgItemInt(hDlg,    ID_EDITH,    (UINT)si.nPos,
FALSE);
}
```

Usamos los valores de la estructura para acotar la posición de la caja y para avanzar y retroceder de página, esto hace que nuestra función sea más independiente y que use menos constantes definidas en el fuente.
También se puede usar el mensaje SBM_GETSCROLLINFO, basta con cambiar la línea:

```
GetScrollInfo(Control, SB_CTL, &si);
```

por esta otra:

```
SendDlgItemMessage(hDlg, ID_SCROLLH,
   SBM_GETSCROLLINFO, 0, (LPARAM)&siv);
```

Devolver valores a la aplicación

Cuando el usuario ha decidido que los valores son los adecuados pulsará el botón de Aceptar. En ese momento deberemos capturar los valores de los controles scroll y actualizar la estructura de parámetros.
También en este caso podemos usar un mensaje o una función para leer la posición del thumb del scrollbar. La función ya la hemos visto un poco más arriba, se trata de GetScrollPos. El mensaje es SBM_GETPOS, ambos devuelven el valor de la posición actual del thumb del control.
Usaremos los dos métodos, uno con cada control. El lugar adecuado para leer esos valores sigue siendo el tratamiento del mensaje WM_COMMAND:

```
     case WM_COMMAND:
           switch(LOWORD(wParam)) {
```

```
            case IDOK:
               Datos->ValorH                        =
GetScrollPos(GetDlgItem(hDlg, ID_SCROLLH), SB_CTL);
               Datos->ValorV                        =
SendDlgItemMessage(hDlg, ID_SCROLLV,
                  SBM_GETPOS, 0, 0);
               EndDialog(hDlg, FALSE);
               return TRUE;
            case IDCANCEL:
               EndDialog(hDlg, FALSE);
               return FALSE;
```

Capítulo 13 Control básico Groupbox

Los GroupBoxes son un estilo de botón que se usa para agrupar controles. Generalmente se usan con controles RadioButton, pero se pueden agrupar controles de cualquier tipo.
El comportamiento es puramente estático, es decir, actúan sólo como marcas y facilitan al usuario el acceso a distintos grupos de controles asociados en función de alguna propiedad común.

Ficheros de recursos

Para comprobar cómo funcionan los groupboxes agruparemos dos conjuntos de controles edit:

```
#include <windows.h>
#include "IDS.h"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 145, 87
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION | WS_SYSMENU
```

```
CAPTION "Group boxes"
FONT 8, "Helv"
BEGIN
 CONTROL "Grupo &1", ID_GROUPBOX1, "BUTTON",
    BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
    8, 4, 64, 65
 CONTROL "Botón 1", ID_BOTON1, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    16, 17, 50, 14
 CONTROL "Botón 2", ID_BOTON2, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    16, 33, 50, 14
 CONTROL "Botón 3", ID_BOTON3, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    16, 49, 50, 14
 CONTROL "Grupo &2", ID_GROUPBOX2, "BUTTON",
    BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
    74, 4, 66, 65
 CONTROL "Botón 4", ID_BOTON4, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    81, 17, 50, 14
 CONTROL "Botón 5", ID_BOTON5, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    81, 33, 50, 14
 CONTROL "Botón 6", ID_BOTON6, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    81, 49, 50, 14
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_GROUP | WS_TABSTOP,
    16, 72, 50, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
```

```
    80, 72, 50, 14
END
```

Hemos añadido los nuevos controles GroupBox. Para más detalles acerca de estos controles ver controles button.
Como se puede observar, un Groupbox no es más que un botón con el estilo BS_GROUPBOX, la principal propiedad de los controles agrupados bajo un groupbox es que es posible moverse a través de ellos usando las teclas del cursor.

```
CONTROL "Grupo &1", ID_GROUPBOX1, "button",
  BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
  8, 4, 64, 65
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los groupbox será el texto que aparecerá en la esquina superior izquierda del control, y que servirá para identificar el grupo.
- id es el identificador del control. El identificador será necesario en algunos casos para vincular los controles entre sí, le veremos más adelante cuando estudiemos los Radio Buttons.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo button y varios de ventana:
  - BS_GROUPBOX: Indica que se trata de un botón con el estilo GroupBox.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_GROUP: marca el control como comienzo de un grupo, el grupo termina cuando empiece el grupo siguiente o terminen los controles.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles GroupBox

Los controles GroupBox no precisan inicialización.

Devolver valores a la aplicación

Tampoco hay ningún valor que retornar desde un control GroupBox.

Capítulo 14 Control básico Checkbox

En realidad no se trata más que de otro estilo de botón.

Normalmente, los CheckBoxes pueden tomar dos valores, encendido y apagado. Aunque también exiten Checkbox de tres estados, en ese caso, el tercer estado corresponde al de inhibido. Los CheckBoxes se usan típicamente para leer opciones que sólo tienen dos posibilidades, del tipo cuya respuesta es sí o no, encendido o apagado, verdadero o falso, etc.
Aunque a menudo se agrupan, en realidad los checkboxes son independientes, cada uno suele tomar un valor de tipo booleano, independientemente de los valores del resto del grupo, si es que pertenece a uno.
El aspecto normal es el de una pequeña caja cuadrada con un texto a uno de los lados, normalmente a la derecha. Cuando está activo se muestra una marca en el interior de la caja, cuando no lo está, la caja aparece vacía. También es posible mostrar el checkbox como un botón corriente, en ese caso, al activarse se quedará pulsado.

Ficheros de recursos

Vamos a mostrar algunos de los posibles aspectos de los CheckBoxes:

```
#include <windows.h>
#include "IDS.h"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 168, 95
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION | WS_SYSMENU
CAPTION "CheckBoxes"
FONT 8, "Helv"
BEGIN
 CONTROL "Normal", ID_NORMAL, "BUTTON",
    BS_CHECKBOX   |   WS_CHILD   |   WS_VISIBLE   |
WS_TABSTOP,
    12, 4, 60, 12
 CONTROL "Auto", ID_AUTO, "BUTTON",
    BS_AUTOCHECKBOX  |  WS_CHILD  |  WS_VISIBLE  |
WS_TABSTOP,
    84, 4, 72, 12
```

```
 CONTROL "Tres estados", ID_TRISTATE, "BUTTON",
    BS_3STATE | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
    12, 17, 60, 12
 CONTROL  "Auto  tres  estados",  ID_AUTOTRISTATE,
"BUTTON",
    BS_AUTO3STATE  |  WS_CHILD  |  WS_VISIBLE  |
WS_TABSTOP,
    84, 16, 76, 12
 CONTROL "Auto Push", ID_AUTOPUSH, "BUTTON",
    BS_AUTOCHECKBOX  |  BS_PUSHLIKE  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    12, 32, 60, 12
 CONTROL  "Auto  Tristate  Push",  ID_AUTOTRIPUSH,
"BUTTON",
    BS_AUTO3STATE  |  BS_PUSHLIKE  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    83, 32, 75, 12
 CONTROL "Derecha", ID_DERECHA, "BUTTON",
    BS_AUTOCHECKBOX  |  BS_LEFTTEXT  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    12, 50, 60, 12
 CONTROL "Plano", ID_PLANO, "BUTTON",
    BS_AUTOCHECKBOX  |  BS_FLAT  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    84, 50, 60, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    12, 69, 50, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON  |  BS_CENTER  |  WS_CHILD  |
WS_VISIBLE | WS_TABSTOP,
    84, 69, 50, 14
END
```

Para ver más detalles acerca de este tipo de controles ver controles button.

Como se puede observar, un CheckBox es un tipo de botón con uno de los siguientes estilos BS_CHECKBOX, BS_AUTOCHECKBOX, BS_3STATE o BS_AUTO3STATE.

Podemos dividir los CheckBoxes en cuatro categorías diferentes, dependiendo de si son o no automáticos o de si son de dos o tres estados.

```
 CONTROL "Normal", ID_NORMAL, "BUTTON",
   BS_CHECKBOX | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
   12, 4, 60, 12
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los CheckBox será el texto que aparecerá acompañando a la caja o en el interior del botón y que servirá para identificar el valor a editar.
- id es el identificador del control. El identificador será necesario en algunos casos para procesar los comandos procedentes del CheckBox, en el caso de los no automáticos será la única forma de tratarlos.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo button y varios de ventana:
  - BS_CHECKBOX: Indica que se trata de un CheckBox de dos estados.
  - BS_AUTOCHECKBOX: Indica que se trata de un CheckBox de dos estados automático.
  - BS_3STATE: Indica que se trata de un CheckBox de tres estados.
  - BS_AUTO3STATE: Indica que se trata de un CheckBox de tres estados automático.
  - BS_PUSHLIKE: Indica que se trata de un CheckBox con la apariencia de un botón corriente.
  - BS_LEFTTEXT: Indica que el texto del CheckBox se sitúa a la izquierda de la caja.
  - BS_FLAT: Indica apariencia plana, sin las sombras que simulan 3D.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_TABSTOP: para que cuando el foco cambie de control al pulsar TAB, pase por este control.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles CheckBox

Los controles CheckBox suelen precisar inicialización.
Para este ejemplo también usaremos variables globales para almacenar los valores de las variables que se pueden editar con los CheckBoxes.

```
// Datos de la aplicación
typedef struct stDatos {
   BOOL Normal;
   BOOL Auto;
   int TriState;
```

```
   int AutoTriState;
   BOOL AutoPush;
   int AutoTriPush;
   BOOL Derecha;
   BOOL Plano;
} DATOS;
```

Creamos una variable estática en nuestro procedimiento de ventana para almacenar los datos, y le daremos valores iniciales a las variables de la aplicación, al procesar el mensaje WM_CREATE:

```
    static DATOS Datos;
    static HINSTANCE hInstance;
...
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           /* Inicialización */
           Datos.Normal = TRUE;
           Datos.Auto = TRUE;
           Datos.TriState = (int)FALSE;
           Datos.AutoTriState = (int)FALSE;
           Datos.AutoPush = FALSE;
           Datos.AutoTriPush = (int)TRUE;
           Datos.Derecha = TRUE;
           Datos.Plano = FALSE;
           return 0;
```

Al crear el cuadro de diálogo, usaremos la función DialogBoxParam, y en el parámetro lParam pasaremos el puntero a nuestra estructura de datos:

```
              DialogBoxParam(hInstance,
"DialogoPrueba", hwnd, DlgProc, (LPARAM)&Datos);
```

Como siempre, para establecer los valores iniciales de los controles CheckBox usaremos el mensaje WM_INITDIALOG del procedimiento de diálogo.
Para eso usaremos la función CheckDlgButton o el mensaje BM_SETCHECK, en este último caso, emplearemos la función SendDlgItemMessage.
De nuevo ilustraremos el ejemplo usando los dos métodos:

```
#define DESHABILITADO -1
...
```

```
    static DATOS *Datos;
...
        case WM_INITDIALOG:
           Datos = (DATOS*)lParam;
           // Estado inicial de los checkbox
           CheckDlgButton(hDlg, ID_NORMAL,
              Datos->Normal    ?    BST_CHECKED    :
BST_UNCHECKED);
           CheckDlgButton(hDlg, ID_AUTO,
              Datos->Auto     ?     BST_CHECKED     :
BST_UNCHECKED);
           if(Datos->TriState != DESHABILITADO)
              CheckDlgButton(hDlg, ID_TRISTATE,
                 Datos->TriState   ?   BST_CHECKED   :
BST_UNCHECKED);
           else
              CheckDlgButton(hDlg,       ID_TRISTATE,
BST_INDETERMINATE);
           if(Datos->AutoTriState != DESHABILITADO)
              CheckDlgButton(hDlg, ID_AUTOTRISTATE,
                 Datos->AutoTriState ? BST_CHECKED :
BST_UNCHECKED);
           else
              CheckDlgButton(hDlg,   ID_AUTOTRISTATE,
BST_INDETERMINATE);
           CheckDlgButton(hDlg, ID_AUTOPUSH,
              Datos->AutoPush    ?    BST_CHECKED    :
BST_UNCHECKED);
           // Usando mensajes:
           if(Datos->AutoTriPush != DESHABILITADO)
              SendDlgItemMessage(hDlg,
ID_AUTOTRIPUSH, BM_SETCHECK,
                 Datos->AutoTriPush                ?
(WPARAM)BST_CHECKED : (WPARAM)BST_UNCHECKED, 0);
           else
              SendDlgItemMessage(hDlg,
ID_AUTOTRIPUSH, BM_SETCHECK,
                 (WPARAM)BST_INDETERMINATE, 0);
           SendDlgItemMessage(hDlg,       ID_DERECHA,
BM_SETCHECK,
```

```
            Datos->Derecha ? (WPARAM)BST_CHECKED :
(WPARAM)BST_UNCHECKED, 0);
         SendDlgItemMessage(hDlg,        ID_PLANO,
BM_SETCHECK,
            Datos->Plano  ?  (WPARAM)BST_CHECKED  :
(WPARAM)BST_UNCHECKED, 0);
         SetFocus(GetDlgItem(hDlg, ID_NORMAL));
         return FALSE;
```

Con esto, el estado inicial de los CheckBoxes será correcto.

Procesar mensajes de los CheckBox

En ciertos casos, será necesario procesar algunos de los mensajes procedentes de los CheckBoxes.

Concretamente, en el caso de los CheckBox no automáticos, su estado no cambia cuando el usuario actúa sobre ellos, sino que será el programa quien deba actualizar ese estado.

Para actualizar el estado de los CheckBoxes cada vez que el usuario actúe sobre ellos debemos procesar los mensajes WM_COMMAND procedentes de ellos.

Quizás, la primera intención sea modificar las variables almacenadas en la estructura Datos para adaptarlas a los nuevos valores. Pero recuerda que es posible que el usuario pulse el botón de "Cancelar". En ese caso, los valores de la estructura Datos no deberían cambiar.

Tenemos dos opciones. Una es usar variables auxiliares para almacenar el estado actual de los CheckBoxes. Otra es leer el estado de los controles directamente. Este último sistema es más seguro, ya que previene el que las variables auxiliares y los controles tengan valores diferentes.

Para leer el estado de los controles tenemos dos posibilidades, como siempre: usar la función IsDlgButtonChecked o el mensaje BM_GETCHECK.

```
      case WM_COMMAND:
         switch(LOWORD(wParam)) {
            case ID_NORMAL:
               if(SendDlgItemMessage(hDlg,
ID_NORMAL, BM_GETCHECK, 0, 0) == BST_CHECKED)
                  SendDlgItemMessage(hDlg,
ID_NORMAL, BM_SETCHECK,
                     (WPARAM)BST_UNCHECKED, 0);
               else
                  SendDlgItemMessage(hDlg,
ID_NORMAL, BM_SETCHECK,
                     (WPARAM)BST_CHECKED, 0);
```

```
                return TRUE;

             case ID_TRISTATE:
                if(IsDlgButtonChecked(hDlg,
ID_TRISTATE) == BST_INDETERMINATE)
                   CheckDlgButton(hDlg,
ID_TRISTATE, BST_UNCHECKED);
                else    if(IsDlgButtonChecked(hDlg,
ID_TRISTATE) == BST_CHECKED)
                   CheckDlgButton(hDlg,
ID_TRISTATE, BST_INDETERMINATE);
                else
                   CheckDlgButton(hDlg,
ID_TRISTATE, BST_CHECKED);
                return TRUE;

           }
```

En este ejemplo intentamos simular el comportamiento de los CheckBoxes automáticos, pero lo normal es que para eso se usen CheckBoxes automáticos. Los no automáticos pueden tener el comportamiento que nosotros prefiramos, en eso consiste su utilidad.

Devolver valores a la aplicación

Por supuesto, lo normal también será que queramos retornar los valores actuales seleccionados en cada CheckBox. A veces también necesitaremos leer el estado de algún control CheckBox durante la ejecución, por ejemplo, cuando eso influye en el estado de otros controles.

Cuando sólo nos interese devolver valores antes de cerrar el diálogo, leeremos esos valores al procesar el mensaje WM_COMMAND para el botón de "Aceptar".

Ya sabemos los dos modos de obtener el estado de los controles CheckBox, la función IsDlgButtonChecked y el mensaje BM_GETCHECK. Ahora sólo tenemos que añadir la lectura de ese estado al procesamiento del mensaje WM_COMMAND del botón "Aceptar".

```
             case IDOK:
                Datos->Normal                    =
(IsDlgButtonChecked(hDlg,       ID_NORMAL)       ==
BST_CHECKED);
                Datos->Auto                      =
(IsDlgButtonChecked(hDlg, ID_AUTO) == BST_CHECKED);
```

```
                if(IsDlgButtonChecked(hDlg,
ID_TRISTATE) == BST_INDETERMINATE)
                   Datos->TriState = DESHABILITADO;
                else
                   Datos->TriState                =
(IsDlgButtonChecked(hDlg,       ID_TRISTATE)       ==
BST_CHECKED);
                if(IsDlgButtonChecked(hDlg,
ID_AUTOTRISTATE) == BST_INDETERMINATE)
                   Datos->AutoTriState            =
DESHABILITADO;
                else
                   Datos->AutoTriState            =
(IsDlgButtonChecked(hDlg,     ID_AUTOTRISTATE)     ==
BST_CHECKED);
                Datos->AutoPush                   =
(IsDlgButtonChecked(hDlg,       ID_AUTOPUSH)       ==
BST_CHECKED);
                if(IsDlgButtonChecked(hDlg,
ID_AUTOTRIPUSH) == BST_INDETERMINATE)
                   Datos->AutoTriPush             =
DESHABILITADO;
                else
                   Datos->AutoTriPush             =
(IsDlgButtonChecked(hDlg,     ID_AUTOTRIPUSH)     ==
BST_CHECKED);
                Datos->Derecha                    =
(IsDlgButtonChecked(hDlg,       ID_DERECHA)       ==
BST_CHECKED);
                Datos->Plano                      =
(IsDlgButtonChecked(hDlg, ID_PLANO) == BST_CHECKED);
                EndDialog(hDlg, FALSE);
                return TRUE;
             case IDCANCEL:
                EndDialog(hDlg, FALSE);
                return FALSE;
```

Y de momento esto es todo lo que diremos sobre CheckBoxes.

Capítulo 15 Control básico RadioButton

De nuevo estamos hablando de un estilo de botón.
Los RadioButtons sólo pueden tomar dos valores, encendido y apagado. Se usan típicamente para leer opciones que sólo tienen un número limitado y pequeño de posibilidades y sólo un valor posible, como por ejemplo: sexo (hombre/mujer), estado civil (soltero/casado/viudo/divorciado), etc.
Es necesario agrupar usando un GroupBox, al menos dos, y con frecuencia tres o más controles de este tipo. No tiene sentido colocar un solo control RadioButton, ya que al menos uno de cada grupo debe estar activo. Tampoco es frecuente agrupar dos, ya que para eso se puede usar un único control CheckBox. Tampoco se agrupan demasiados, ya que ocupan mucho espacio, en esos casos es mejor usar un ComboBox o un ListBox.
El aspecto normal es el de un pequeño círculo con un texto a uno de los lados, normalmente a la derecha. Cuando está activo se muestra el círculo relleno, cuando no lo está, aparece vacío. También es posible mostrar el RadioButton como un botón corriente, en ese caso, al activarse se quedará pulsado.

Ficheros de recursos

Vamos a mostrar algunos de los posibles aspectos de los RadioButtons:

```
#include <windows.h>
#include "IDS.h"

Menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "&Diálogo", CM_DIALOGO
 END
END

DialogoPrueba DIALOG 0, 0, 179, 89
STYLE  DS_MODALFRAME  |  WS_POPUP  |  WS_VISIBLE  |
WS_CAPTION | WS_SYSMENU
CAPTION "RadioButtons"
FONT 8, "Helv"
BEGIN
 CONTROL "Grupo 1", ID_GRUPO1, "BUTTON",
    BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
    4, 5, 76, 52
```

```
 CONTROL "RadioButton 1", ID_RADIOBUTTON1, "BUTTON",
    BS_AUTORADIOBUTTON  |  WS_CHILD  |  WS_VISIBLE  |
WS_TABSTOP,
    11, 15, 60, 12
 CONTROL "RadioButton 2", ID_RADIOBUTTON2, "BUTTON",
    BS_AUTORADIOBUTTON  |  WS_CHILD  |  WS_VISIBLE  |
WS_TABSTOP,
    11, 28, 60, 12
 CONTROL "RadioButton 3", ID_RADIOBUTTON3, "BUTTON",
    BS_AUTORADIOBUTTON  |  WS_CHILD  |  WS_VISIBLE  |
WS_TABSTOP,
    11, 41, 60, 12
 CONTROL "Grupo 2", ID_GRUPO2, "BUTTON",
    BS_GROUPBOX | WS_CHILD | WS_VISIBLE | WS_GROUP,
    89, 5, 76, 52
 CONTROL "RadioButton 4", ID_RADIOBUTTON4, "BUTTON",
    BS_RADIOBUTTON   |   BS_PUSHLIKE   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    96, 15, 60, 12
 CONTROL "RadioButton 5", ID_RADIOBUTTON5, "BUTTON",
    BS_RADIOBUTTON   |   BS_PUSHLIKE   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    96, 28, 60, 12
 CONTROL "RadioButton 6", ID_RADIOBUTTON6, "BUTTON",
    BS_RADIOBUTTON   |   BS_PUSHLIKE   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    96, 41, 60, 12
 CONTROL "Aceptar", IDOK, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    8, 69, 50, 14
 CONTROL "Cancelar", IDCANCEL, "BUTTON",
    BS_PUSHBUTTON   |   BS_CENTER   |   WS_CHILD   |
WS_VISIBLE | WS_TABSTOP,
    68, 69, 50, 14
END
```

Para ver más detalles acerca de este tipo de controles ver controles button.
Como se puede observar, un RadioButton es un tipo de botón con uno de los siguientes estilos BS_RADIOBUTTON o BS_AUTORADIOBUTTON.

También se puede ver que hemos agrupado los controles RadioButton en dos grupos de tres botones. Cada grupo empieza con un control GroupBox con el estilo WS_GROUP. Más adelante veremos cómo trabajan en conjunto los GroupBoxes y los RadioButtons.

```
 CONTROL   "Normal",   ID_RADIOBUTTON1,   "BUTTON",
BS_RADIOBUTTON | WS_CHILD | WS_VISIBLE | WS_TABSTOP,
12, 4, 60, 12
```

- CONTROL es la palabra clave que indica que vamos a definir un control.
- A continuación, en el parámetro text, en el caso de los RadioButtons será el texto que aparecerá acompañando al círculo o en el interior del botón y que servirá para identificar el valor a elegir.
- id es el identificador del control. El identificador será necesario en algunos casos para procesar los comandos procedentes del RadioButton, en el caso de los no automáticos será la única forma de tratarlos.
- class es la clase de control, en nuestro caso "BUTTON".
- style es el estilo de control que queremos. En nuestro caso es una combinación de un estilo button y varios de ventana:
  - BS_RADIOBUTTON: Indica que se trata de un RadioButton no automático.
  - BS_AUTORADIOBUTTON: Indica que se trata de un RadioButton automático.
  - BS_PUSHLIKE: Indica que se trata de un RadioButton con la apariencia de un botón corriente.
  - BS_LEFTTEXT: Indica que el texto del RadioButton se sitúa a la izquierda de la caja.
  - BS_FLAT: Indica apariencia plana, sin las sombras que simulan 3D.
  - WS_CHILD: crea el control como una ventana hija.
  - WS_VISIBLE: crea una ventana inicialmente visible.
  - WS_TABSTOP: para que cuando el foco cambie de control al pulsar TAB, pase por este control.
- coordenada x del control.
- coordenada y del control.
- width: anchura del control.
- height: altura del control.

Iniciar controles RadioButton

Los controles RadioButton necesitan ser inicializados. Para este ejemplo también usaremos variables globales para almacenar los valores de las variables que se pueden editar con los RadioButtons. En general se necesita una única variable para cada grupo de RadioButtons.

```
// Datos de la aplicación
typedef struct stDatos {
   int Grupo1;
```

```
   int Grupo2;
} DATOS;
```

Crearemos una estructura de datos estática en nuestro procedimiento de ventana, y le daremos valores iniciales al procesar el mensaje WM_CREATE:

```
    static DATOS Datos;
...
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           /* Inicialización */
           Datos.Grupo1 = 1;
           Datos.Grupo2 = 2;
           return 0;
```

Finalmente, usaremos la función DialogBoxParam para crear el diálogo, y pasaremos en el parámetro lParam un puntero a la estructura de datos.

```
              DialogBoxParam(hInstance,
"DialogoPrueba", hwnd, DlgProc, (LPARAM)&Datos);
```

Como siempre, para establecer los valores iniciales de los controles CheckBox usaremos el mensaje WM_INITDIALOG del procedimiento de diálogo.
Para eso usaremos la función CheckRadioButton o el mensaje BM_SETCHECK, en este último caso, emplearemos la función SendDlgItemMessage. Usar el mensaje implica enviar un mensaje al menos a dos controles RadioButton del grupo, el que se activa y el que se desactiva.
De nuevo ilustraremos el ejemplo usando los dos métodos:

```
    static DATOS *Datos;
...
        case WM_INITDIALOG:
           Datos = (DATOS*)lParam;
           // Estado inicial de los radiobuttons
           CheckRadioButton(hDlg,    ID_RADIOBUTTON1,
ID_RADIOBUTTON3,
              ID_RADIOBUTTON1+Datos->Grupo1-1);
           // Usando mensajes:
           SendDlgItemMessage(hDlg, ID_RADIOBUTTON4,
              BM_SETCHECK,     (WPARAM)BST_UNCHECKED,
0);
```

```
            SendDlgItemMessage(hDlg, ID_RADIOBUTTON5,
               BM_SETCHECK,     (WPARAM)BST_UNCHECKED,
0);
            SendDlgItemMessage(hDlg, ID_RADIOBUTTON6,
               BM_SETCHECK,     (WPARAM)BST_UNCHECKED,
0);
            SendDlgItemMessage(hDlg,
ID_RADIOBUTTON4+Datos->Grupo2-1,
               BM_SETCHECK, (WPARAM)BST_CHECKED, 0);
            SetFocus(GetDlgItem(hDlg,
ID_RADIOBUTTON1));
            return FALSE;
```

Es muy importante asignar identificadores correlativos a los controles de cada grupo. Esto nos permite por una parte usar el mensaje CheckRadioButton, tanto como expresiones como ID_RADIOBUTTON1+Datos>Grupo1-1. (Si hubierámos empezado por cero para el primer control del grupo, no sería necesario restar uno).
Usando los mensajes nos vemos obligados a quitar la marca a todos los controles del grupo 2. Esto es porque desconocemos el estado inicial de los controles. En este caso es mucho mejor usar la función que el mensaje.
Con esto, el estado inicial de los RadioButtons será correcto.

Procesar mensajes de los RadioButtons

En ciertos casos, será necesario procesar algunos de los mensajes procedentes de los RadioButtons.
Concretamente, en el caso de los RadioButtons no automáticos, su estado no cambia cuando el usuario actúa sobre ellos, sino que será el programa quien deba actualizar ese estado.
Para actualizar el estado de los RadioButtons cada vez que el usuario actúe sobre ellos debemos procesar los mensajes WM_COMMAND procedentes de ellos.
Como sucedía con los CheckBoxes, la primera intención puede que sea modificar las variables almacenadas en la estructura Datos para adaptarlas a los nuevos valores. Pero recuerda que es posible que el usuario pulse el botón de "Cancelar". En ese caso, los valores de la estructura Datos no deberían cambiar.
De nuevo tenemos dos opciones. Una es usar variables auxiliares para almacenar el estado actual de los RadioButtons. Otra es leer el estado de los controles directamente. Este último sistema es más seguro, ya que previene el que las variables auxiliares y los controles tengan valores diferentes.
Para leer el estado de los controles tenemos dos posibilidades, como siempre: usar la función IsDlgButtonChecked o el mensaje BM_GETCHECK. Veremos las dos opciones.
Con funciones:

```
            switch(LOWORD(wParam)) {
               case ID_RADIOBUTTON4:
               case ID_RADIOBUTTON5:
               case ID_RADIOBUTTON6:
                  CheckRadioButton(hDlg,
ID_RADIOBUTTON4, ID_RADIOBUTTON6,
                     LOWORD(wParam));
                  return TRUE;
```

Con mensajes:

```
            switch(LOWORD(wParam)) {
               case ID_RADIOBUTTON4:
               case ID_RADIOBUTTON5:
               case ID_RADIOBUTTON6:
                  SendDlgItemMessage(hDlg,
ID_RADIOBUTTON4, BM_SETCHECK,
                     (WPARAM)BST_UNCHECKED, 0);
                  SendDlgItemMessage(hDlg,
ID_RADIOBUTTON5, BM_SETCHECK,
                     (WPARAM)BST_UNCHECKED, 0);
                  SendDlgItemMessage(hDlg,
ID_RADIOBUTTON6, BM_SETCHECK,
                     (WPARAM)BST_UNCHECKED, 0);
                  SendDlgItemMessage(hDlg,
LOWORD(wParam), BM_SETCHECK,
                     (WPARAM)BST_CHECKED, 0);
                  return TRUE;
```

En este ejemplo intentamos simular el comportamiento de los RadioButtons automáticos, pero lo normal es que para eso se usen RadioButtons automáticos. Los no automáticos pueden tener el comportamiento que nosotros prefiramos, y en eso consiste su utilidad. Los automáticos no requieren ninguna atención de nuestro programa.

Devolver valores a la aplicación

Por supuesto, lo normal también será que queramos retornar los valores actuales seleccionados en cada Grupo de RadioButtons.
Cuando nos interese devolver valores antes de cerrar el diálogo, leeremos esos valores al procesar el mensaje WM_COMMAND para el botón de "Aceptar".
Ya sabemos los dos modos de obtener el estado de los controles RadioButtons, la función IsDlgButtonChecked y el mensaje BM_GETCHECK. Ahora sólo tenemos

que añadir la lectura de ese estado al procesamiento del mensaje WM_COMMAND del botón "Aceptar".

```
               case IDOK:
                  if(IsDlgButtonChecked(hDlg,
ID_RADIOBUTTON1) == BST_CHECKED)
                     Datos->Grupo1 = 1;
                  else
                  if(IsDlgButtonChecked(hDlg,
ID_RADIOBUTTON2) == BST_CHECKED)
                     Datos->Grupo1 = 2;
                  else
                     Datos->Grupo1 = 3;
                  if(SendDlgItemMessage(hDlg,
ID_RADIOBUTTON4,
                     BM_GETCHECK,     0,     0)     ==
BST_CHECKED) Datos->Grupo2 = 1;
                  else
                  if(SendDlgItemMessage(hDlg,
ID_RADIOBUTTON5,
                     BM_GETCHECK,     0,     0)     ==
BST_CHECKED) Datos->Grupo2 = 2;
                  else
Datos->Grupo2 = 3;
                  EndDialog(hDlg, FALSE);
                  return TRUE;
               case IDCANCEL:
                  EndDialog(hDlg, FALSE);
                  return FALSE;
```

Con esto queda cerrado de momento el tema de los RadioButtons.

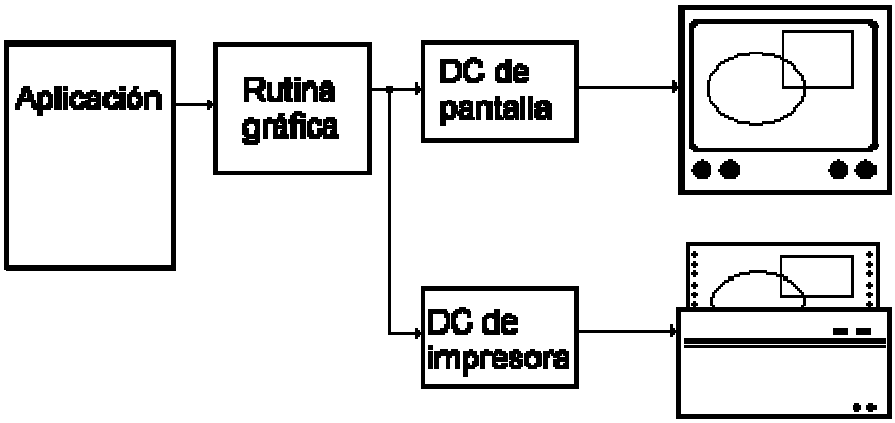
Capítulo 16 El GDI

Ahora ya dominamos los controles básicos, así que pasaremos a un capítulo muy amplio, pero mucho más interesante.

El GDI (Graphics Device Interface), es el interfaz de dispositivo gráfico, que contiene todas las funciones y estructuras necesarias que nos permiten comunicar nuestras aplicaciones con cualquier dispositivo gráfico de salida conectado a nuestro ordenador: pantalla, impresora, plotter, etc.

Veremos que podremos trazar líneas, curvas, figuras cerradas, polígonos y mapas de bits. Además, podremos controlar características como colores, aspectos de líneas y tramas de superficies rellenas, mediante objetos como plumas, pinceles y fuentes de caracteres.

Las aplicaciones Windows dirigen las salidas gráficas a lo que se conoce como un Contexto de Dispositivo, abreviado como DC. Cada DC se crea para un dispositivo concreto.
Un DC es una de las estructuras del GDI, que contiene información sobre el dispositivo, como las opciones seleccionadas, o modos de funcionamiento.



La aplicación crea un DC mediante una función del GDI, y éste devuelve un manipulador de DC (hDC), que se usará en las siguientes llamadas para identificar el dispositivo que recibirá la salida. Mediante el DC, la aplicación también puede obtener ciertas capacidades del dispositivo, como las dimensiones del área de impresión, la resolución, etc.
La salida puede enviarse directamente al DC del dispositivo físico, o bien a un dispositivo lógico, que se crea en memoria o en un metafile. La ventaja de usar un dispositivo lógico es que se almacena la salida completa y posteriormente puede enviarse al cualquier dispositivo físico.
También existen funciones para seleccionar diferentes modos y opciones del dispositivo. Eso incluye colores del texto y fondo, plumas y pinceles de diferentes colores y texturas para trazar líneas o rellenar superficies, y lo que se conoce como operaciones binarias de patrones (Binary Raster Operations), que indican cómo se combinan las nuevas salidas gráficas con las existentes previamente. También existen varios sistemas de mapeo de coordenadas, para traducir las coordenadas usadas en la aplicación con las el sistema de coordenadas del dispositivo.

Objetos del GDI

Windows usa objetos y manipuladores para acceder a los recursos del sistema, en lo que se refiere al GDI existen los siguientes objetos:
Mapas de bits, pinceles, fuentes, plumas, plumas extendidas, regiones, contextos de dispositivos, contextos de dispositivo de memoria, metafiles, contextos de dispositivo de metafiles, metafiles mejorados y contextos de dispositivo de metafiles mejorados.
Veremos cada uno de estos objetos, algunos en los próximos capítulos, los que denominaremos básicos: mapas de bits, pinceles, fuentes, plumas y dispositivos de contexto. El resto en capítulos más avanzados.

El API proporciona funciones para crear objetos y manipuladores de objetos, cerrar un manipulador de objeto y destruir un objeto.
Los objetos y los manipuladores consumen memoria, es muy importante cerrar los manipuladores y destruir los objetos cuando no se necesiten más, no hacerlo puede ralentizar el sistema e incluso volverlo inestable.

Capítulo 17 Objetos básicos del GDI: El Contexto de dispositivo, DC

La estructura del DC, como la del resto de los objetos del GDI, no es accesible directamente desde las aplicaciones. Contiene los valores de ciertos atributos que determinan la posición y apariencia de la salida hacia un dispositivo.
Cada vez que se crea un DC, cada atributo tiene un valor por defecto conocido. Los atributos almacenados en el DC, y sus valores por defecto son:

Atributo	Valor por defecto
Color de fondo	Blanco (Depende del seleccionado en el panel de control)
Modo del fondo	Opaco
Mapa de bits (bitmap)	Ninguno
Pincel (Brush)	WHITE_BRUSH (Blanco)
Origen para el pincel	(0,0)
Región de recorte (Clipping region)	Toda la superficie visible de la ventana
Paleta de colores	DEFAULT_PALETTE (paleta por defecto)
Posición actual de la pluma	(0,0)
Origen del dispositivo	Esquina superior derecha de la ventana o del área de cliente.
Modo de trazado	R2_COPYPEN (sustitución del valor actual en pantalla)
Fuente	SYSTEM_FONT (fuente del sistema)
Espacio entre caracteres	0
Modo de mapeo	MM_TEXT (coordenadas de texto)
Pluma	BLACK_PEN (negra)
Modo para el relleno de polígonos	ALTERNATE (alternativo)
Modo de estiramiento (Stretching mode)	BLACKONWHITE
Color de texto	Negro
Tamaño del Viewport	(1,1)

Origen del Viewport	(0,0)
Tamaño de la ventana	(1,1)
Origen de la ventana	(0,0)

Existen funciones para acceder a los valores del DC, mediante un manipulador, ya sea para leer sus valores o para modificarlos. En próximos capítulos veremos algunas de ellas.

Actualizar el área de cliente de una ventana, el mensaje WM_PAINT

Todos los ejemplos estarán centrados en el proceso del mensaje WM_PAINT, que le indica al procedimiento de ventana que parte o toda su superficie debe ser actualizada.

En nuestros primeros ejemplos y mientras veamos los diferentes objetos, funciones y estructuras, no necesitaremos crear un DC, simplemente obtendremos un manipulador para el DC de la ventana de nuestro programa. Cada ventana tiene asociados varios DCs. A nosotros nos interesa el que está asociado al área de cliente.

Para actualizar el área de cliente de nuestra ventana tendremos que seguir una especie de "ritual". Eso es imprescindible, y a menudo se siguen estas "fórmulas" en muchos procesos de los programas Windows, muchas veces sin saber el motivo de cada paso. Pero en este curso intentaremos explicar el por qué de cada uno de los pasos, y que acciones desempeña cada uno de ellos.

Lo que sigue es un ejemplo muy sencillo de cómo se procesa un mensaje WM_PAINT:

```
    HDC hDC;
    PAINTSTRUCT ps;
...
        case WM_PAINT:
           hDC = BeginPaint(hwnd, &ps);
           Ellipse(hDC, 10, 10, 50, 50);
           EndPaint(hwnd, &ps);
           break;
```

Hay más formas de enviar salidas a una ventana, y las veremos más adelante, pero la más sencilla es ésta.

Utilizaremos la función BeginPaint para poder obtener un DC a nuestra ventana. Esta función además ajusta el área de "clipping", eso sirve para que sólo se actualice la parte de la ventana que sea necesario. Por ejemplo, imaginemos que nuestra ventana está parcialmente oculta por otras, y que la pasamos a primer plano. En ese caso, el área de "clipping" será sólo la correspondiente a las partes que estaban ocultas previamente, el resto de la ventana no se actualizará. Esto

ahorra bastante tiempo de proceso, ya que dibujar gráficos suele ser un proceso lento.
El segundo parámetro es un puntero a una estructura PAINTSTRUCT que es necesario para la llamada. Los datos obtenidos mediante esa estructura pueden sernos útiles en algunas ocasiones, pero de momento sólo la usaremos para obtener un DC del área de cliente de la ventana.
Una vez que hemos obtenido un DC para nuestra ventana, podremos utilizarlo para dibujar en ella. En este ejemplo sólo se dibuja una circunferencia, usando al función Ellipse. En próximos ejemplos iremos complicando más esa parte.
Por último, una vez que hemos terminado de pintar, liberamos el DC usando la función EndPaint. Es muy importante hacer esto cada vez que obtengamos un DC mediante **BeginPaint**.

Colores

Windows almacena los colores en un entero de 32 bits en formato RGB, hay un tipo definido para eso, se llama COLORREF. Los 8 bits de menor peso se usan para almacenar la componente de rojo, los 8 siguientes para el verde, los 8 siguientes para el azul y el resto no se usa, y debe ser cero.
Para referenciar o asignar un color se usa la macro RGB, que admite tres parámetros de tipo BYTE, cada uno de ellos indica la intensidad relativa de una de las componentes: rojo, verde o azul, y devuelve un COLORREF.
También es posible obtener cada una de las componentes de un COLORREF mediante las macros GetRValue, GetGValue y GetBValue.

Capítulo 18 Objetos básicos del GDI:
La pluma (Pen)

La pluma se utiliza para trazar líneas y curvas. En este capítulo veremos cómo crearlas, seleccionarlas y destruirlas, y cómo elegir el estilo, grosor y color para una pluma.
El proceso con los objetos es siempre el mismo, hay que crearlos, seleccionarlos y, cuando ya no se necesiten, destruirlos.

Plumas de Stock

La única excepción a lo dicho antes es un pequeño almacén de objetos que usa el sistema y que están disponibles para usar en nuestros programas.
El ejemplo anterior funciona gracias a ello, ya que usa la pluma por defecto del DC. En el caso de los objetos de stock, no es necesario crearlos ni destruirlos (aunque esto último no está prohibido), siempre podemos obtener un manipulador y seleccionarlo para usarlo.
Concretamente, en el caso de las plumas, existen tres en el stock:

Valor	Significado

BLACK_PEN	Pluma negra
NULL_PEN	Pluma nula
WHITE_PEN	Pluma blanca

Para obtener un manipulador para una de esos objetos de stock se usa la función GetStockObject. Dependiendo del valor que usemos obtendremos un tipo de objeto diferente.

Plumas cosméticas y geométricas

Existen dos categorías de plumas.
Las cosméticas se crean con un grosor expresado en unidades de dispositivo, es decir, las líneas que se tracen con estas plumas tendrán siempre el mismo grosor. Estas plumas sólo tienen tres atributos: grosor, color y estilo. Las plumas de stock son de este tipo.
Las geométricas se crean con un grosor expresado en unidades lógicas. Esto significa que el grosor de las líneas puede ser escalado, y depende de la transformaciones actuales del "mundo actual" (veremos esto en siguientes capítulos). Además de los tres atributos que también tienen las plumas cosméticas, las geométricas poseen otros cuatro: plantilla, trama opcional, estilos para extremos y para uniones.
Las líneas trazadas con plumas cosméticas son entre tres y diez veces más rápidas que las de las geométricas.

Crear una pluma

Si no vamos a utilizar una pluma de stock, (la verdad es que están muy limitadas), podemos crear nuestras propias plumas.
Para ello disponemos de la función CreatePen. Esta función también nos devuelve un manipulador, en este caso de pluma, que podremos usar posteriormente para seleccionar la nueva pluma.
CreatePen nos pide tres parámetros. El primero es el estilo de línea, podemos elegir uno de los siguientes valores:

Estilo	Descripción
PS_SOLID	Las líneas serán continuas y sólidas.
PS_DASH	Líneas de trazos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.
PS_DOT	Líneas de puntos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.
PS_DASHDOT	Líneas alternan puntos y trazos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.
PS_DASHDOTDOT	Líneas alternan líneas y dobles puntos. Este estilo sólo es válido cuando el ancho de la pluma sea uno o menos en unidades de dispositivo.
PS_NULL	Las líneas son invisibles.

PS_INSIDEFRAME	Las líneas serán sólidas. Cuando ésta pluma se usa en cualquier función de dibujo del GDI que requiera un rectángulo que sirva como límite, las dimensiones de la figura se reducirán para que se ajusten por completo al interior del rectángulo, teniendo en cuenta el grosor de la pluma. Esto sólo se aplica a plumas geométricas.

El segundo parámetro es el grosor de la línea en unidades lógicas, si se especifica un grosor de cero, la línea tendrá un pixel de ancho.
El último parámetro es un COLORREF, que será el color de las líneas.
Evidentemente, CreatePen sólo puede crear plumas cosméticas.
Existen otras dos funciones para crear plumas: CreatePenIndirect y ExtCreatePen.
La primera sirve también para crear plumas cosméticas, pero lo hace a través de una estructura LOGPEN, que almacena en su interior los mismos parámetros que se pasan a la función CreatePen. Esto puede ser útil cuando creemos muchas plumas distintas o varias con muy pocas diferencias.
La segunda nos permite crear tanto plumas cosméticas como geométricas. De momento no veremos este tipo de plumas, las dejaremos para capítulos más avanzados.

Seleccionar una pluma

Aunque podemos disponer de un repertorio de manipuladores de pluma, sólo puede haber una activa en cada momento, para seleccionar la pluma activa se usa la función SelectObject.
En realidad, ésta función sirve para seleccionar cualquier tipo de objeto, no sólo plumas. El tipo de objeto seleccionado depende el parámetro que se pase a la función. El nuevo objeto seleccionado reemplaza al actual, y se devuelve el manipulador del objeto seleccionado anteriormente.
Se debe guardar el manipulador de la pluma por defecto seleccionada antes de cambiarla por primera vez, y restablecerlo antes de terminar el procedimiento de pintar.

Destruir una pluma

Por último, cuando ya no necesitemos más los manipuladores de plumas, debemos destruirlos, con el fin de liberar la memoria usada para almacenarlos. Esto se hace mediante la función DeleteObject.

Capítulo 19 Funciones para el trazado de líneas

El GDI dispone de un repertorio de funciones para el trazado de líneas bastante completo

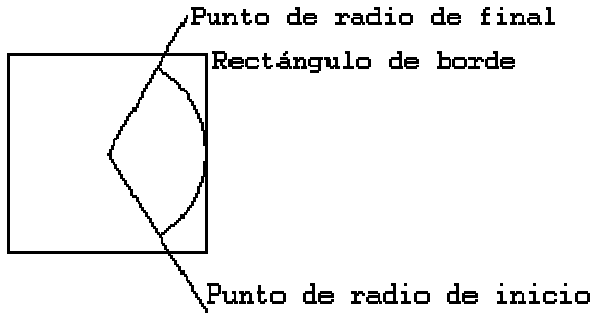
Función	Tipo de línea
MoveToEx	Actualiza la posición actual del cursor gráfico, opcionalmente obtiene la posición anterior.
LineTo	Traza una línea desde la posición actual del cursor al punto indicado.

ArcTo	Traza un arco de elipse.
PolylineTo	Traza uno o más trazos de líneas rectas.
PolyBezierTo	Traza una o más curvas Bézier.
AngleArc	Traza un segmento de arco de circunferencia.
Arc	Traza un arco de elipse.
Polyline	Traza una serie de segmentos de recta que conectan los puntos de un array.
PolyBezier	Traza una o más curvas Bézier.
GetArcDirection	Devuelve la dirección de arco del DC actual.
SetArcDirection	Cambia la dirección del ángulo para el trazado de arcos y rectángulos.
LineDDA	Traza una línea, pero permite a una función de usuario decidir qué pixels se mostrarán.
LineDDAProc	Función callback de la aplicación que procesa las coordenadas recibidas de LineDDA.
PolyDraw	Traza una o varias series de líneas y curvas Bézier.
PolyPolyLine	Traza una o varias series de segmentos de recta conectados.

La mayoría de éstas funciones no requieren mayor explicación, sobre todo las que trazan arcos y líneas rectas.

Trazado de arcos, función Arc

Para trazar un arco mediante la función Arc, se parte de una circunferencia inscrita en el rectángulo de borde. Los puntos de inicio y final del arco se obtienen de los cortes de los radios definidos por los puntos de radio de inicio y final:

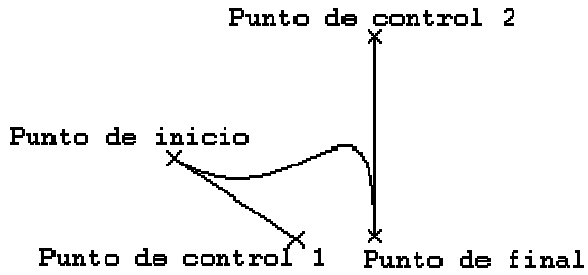


La función ArcTo es idéntica, salvo que se traza una línea desde la posición actual del cursor gráfico hasta el punto de inicio del arco.

Curvas Bézier

Las curvas Bézier son una forma de definir curvas irregulares mediante métodos matemáticos. Para definir una curva Bézier sólo se necesitan cuatro puntos: los puntos de inicio y final, y dos puntos de control.
En el gráfico se representa una curva Bézier y las líneas auxiliares, que sólo se muestran como ayuda. El punto de inicio y el punto de control 1 refinen una recta

que es tangente a la curva en el punto de inicio. La curva tiende a aproximarse al punto de control 1.
Análogamente, el punto de final y el punto de control 2 refinen otra recta, que es tangente a la curva en el punto de final. La curva también tiende a acercarse al punto de control 2.
Las ecuaciones que definen la curva no son demasiado complejas, pero escapan al objetivo de este curso, se estudiaran las curvas de Bézier con detalle en un artículo separado.



Funciones Poly<tipo>

Todas las funciones con el prefijo Poly trabajan de un modo parecido. Se basan en un array de puntos para definir un conjunto de líneas que se trazan una a continuación de otra.
Polyline o PolylineTo trazan un conjunto de segmentos rectos, PolyBezier y PolyBezierTo, un conjunto de curvas Bézier. PolyDraw, varios conjuntos de líneas y curvas Bézier, y PolyPolyline, varios conjuntos de líneas rectas.

Función LineDDA y funciones callback LineDDAProc

LineDDA nos permite personalizar el trazado de segmentos de líneas rectas. Mediante la definición de funciones propias del tipo LineDDAProc, podemos procesar cada punto de la línea y decidir cómo visualizarlo. Podemos cambiar el color, ignorar ciertos puntos, y en general, aplicar la modificación que queramos. Esto nos permite trazar líneas de varios colores o con distintas tramas, etc.
Veamos un ejemplo sencillo, definiremos una función LineDDAProc para trazar líneas que alternen 10 pixels rojos y 10 verdes.

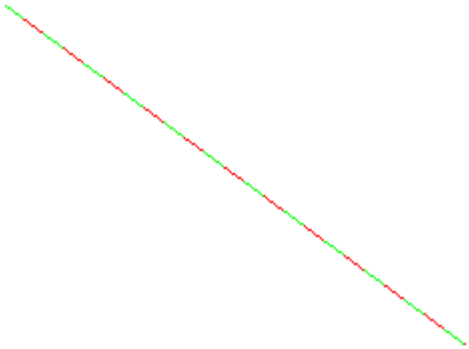
```
struct DatosDDA1 {
   int cuenta;
   HDC hdc;
};

VOID CALLBACK FuncionDDA1(int X, int Y, LPARAM
datos)
{
```

```
   struct DatosDDA1 *dato = (struct DatosDDA1
*)datos;

   // Función que pinta líneas con 10 pixels
alternados rojos y verdes
   dato->cuenta++;
   if(dato->cuenta >= 20) dato->cuenta = 0; //
Mantenemos cuenta entre 0 y 19
   if(dato->cuenta < 10)
     SetPixel(dato->hdc, X, Y, RGB(0,255,0));
   else
     SetPixel(dato->hdc, X, Y, RGB(255,0,0));
}
```

Hemos definido una función callback para que muestre cada punto en función de los datos almacenados en una estructura DatosDDA1, diseñada por nosotros. En ella almacenamos un valor "cuenta" que nos ayuda a decidir el color de cada pixel, y un manipulador de DC para poder activar pixels, usando SetPixel, en la ventana:

```
...
   struct DatosDDA1 datos = {0, hDC};

   LineDDA(10,10,       240,180,       FuncionDDA1,
(LPARAM)&datos);
...
```

Ahora podemos llamar a la función LineDDA, indicando los puntos de inicio y final de la línea, la función que usaremos para decidir el color de cada punto, y un puntero a la estructura de datos que esa función necesita.



Capítulo 20 Objetos básicos del GDI: El pincel (Brush)

El pincel se utiliza para rellenar superficies y figuras cerradas. En este capítulo veremos cómo crearlos, seleccionarlos y destruirlos, y cómo elegir el estilo, textura y color para un pincel.

Al igual que vimos con las plumas, el proceso con todos los objetos es siempre el mismo, hay que crearlos, seleccionarlos y, cuando ya no se necesiten, destruirlos.

Pinceles lógicos

Existen cuatro tipos distintos de pinceles lógicos.

- Sólidos, consisten en un único color continuo.
- Stock, pinceles predefinidos por el sistema.
- Hatch, tramas de líneas.
- Patrones, consisten en mapas de bits.

Pinceles sólidos

Son los más simples, se usan para rellenar superficies con un único color uniforme. Para crear uno de estos pinceles se usa la función CreateSolidBrush, que sólo requiere que se especifique el color del pincel.

Pinceles de Stock

Del mismo modo que sucede con las plumas, también disponemos de un juego de pinceles de stock que podremos usar en nuestros programas.

En el caso de los objetos de stock, no es necesario crearlos ni destruirlos, siempre podemos obtener un manipulador y seleccionarlo para usarlo.

En el caso de los pinceles, existen siete en el stock:

Valor	Significado
BLACK_BRUSH	Pincel negra
DKGRAY_BRUSH	Pincel gris oscuro
GRAY_BRUSH	Pincel gris
HOLLOW_BRUSH	Pincel hueco
LTGRAY_BRUSH	Pincel gris claro
NULL_BRUSH	Pincel nulo (equivale a HOLLOW_BRUSH)
WHITE_BRUSH	Pincel blanco

Ya vimos que para obtener un manipulador para una de esos objetos de stock se usa la función GetStockObject. Dependiendo del valor que usemos obtendremos un tipo de objeto diferente.

Pinceles de tramas (Hatch)

Consisten en tramas de líneas paralelas, que permiten crear superficies ralladas.

Para crear pinceles tramados se usa la función CreateHatchBrush. Además del color podemos escoger entre varias tramas distintas.
Existen seis tipos de tramas predefinidas, accesibles mediante constantes:

Valor	Significado
HS_BDIAGONAL	Trama de líneas diagonales a 45º descendentes de izquierda a derecha.
HS_CROSS	Trama de líneas horizontales y verticales.
HS_DIAGCROSS	Trama de líneas diagonales a 45º cruzadas.
HS_FDIAGONAL	Trama de líneas diagonales a 45º ascendentes de izquierda a derecha.
HS_HORIZONTAL	Trama de líneas horizontales.
HS_VERTICAL	Trama de líneas verticales.

Pinceles de patrones

Este tipo de pincel se crea a partir de un mapa de bits, así que en realidad puede ser cualquier tipo de imagen. Antes de poder crear uno de estos pinceles tendremos que disponer de un mapa de bits, que, como comentamos antes, es otro de los objetos de GDI que podemos manejar.
Supondremos que ya sabemos crear un objeto de mapa de bits, aunque explicaremos cómo hacerlo en próximos capítulos.
Para crear un pincel de patrón disponemos de tres funciones: CreatePatternBrush, CreateDIBPatternBrushPt y CreateDIBPatternBrush. La primera sólo requiere un manipulador de un mapa de bits. La segunda y la tercera permiten usar mapas de bits independientes del dispositivo, lo cual proporciona mayor control sobre la paleta de colores.
Para poder usar este tipo de pinceles hay que estar algo más familiarizado con los mapas de bits, volveremos sobre este tema cuando hayamos visto el objeto bitmap.

Crear un pincel

Además de las funciones mencionadas para crear pinceles de los distintos tipos lógicos mencionados, existe una función más para crear pinceles: CreateBrushIndirect.
Esta función sirve para crear pinceles lógicos, pero lo hace a través de una estructura LOGBRUSH, que almacena en su interior los parámetros necesarios para crear un pincel sólido, de trama o de patrón.

Seleccionar un pincel

Con los pinceles sucede lo mismo que con las plumas, aunque podemos tener de un repertorio de manipuladores de pincel, sólo puede haber uno activo en cada momento, para seleccionar el pincel activo se usa la función SelectObject.
Como ya hemos dicho el tipo de objeto seleccionado depende del parámetro que se pase a la función. El nuevo objeto seleccionado reemplaza al actual, y se devuelve el manipulador del objeto seleccionado anteriormente.

Se debe guardar el manipulador del pincel por defecto seleccionado antes de cambiarlo por primera vez, y restablecerlo antes de terminar el procedimiento de pintar.

Destruir un pincel

Por último, cuando ya no necesitemos más los manipuladores de pinceles, debemos destruirlos, con el fin de liberar la memoria usada para almacenarlos. Esto se hace mediante la función DeleteObject.

Capítulo 21 Funciones para el trazado de figuras rellenas

Veremos ahora el repertorio de funciones para el trazado de figuras cerradas rellenas. Para trazar estas figuras se usa una pluma para el borde y un pincel para los interiores.

Función	Tipo de figura
Chord	Traza una figura definida por el corte de una elipse y una recta secante y la rellena.
Ellipse	Traza una elipse rellena.
FillRect	Rellena un rectángulo, sin trazar el borde.
FrameRect	Traza un borde alrededor de un rectángulo usando el pincel actual.
Pie	Traza un sector de elipse.
Polygon	Traza un polígono relleno.
PolyPolygon	Traza una serie de polígonos cerrados y rellenos.
Rectangle	Traza un rectángulo relleno.
RoundRect	Traza un rectángulo relleno con las esquinas redondeadas.

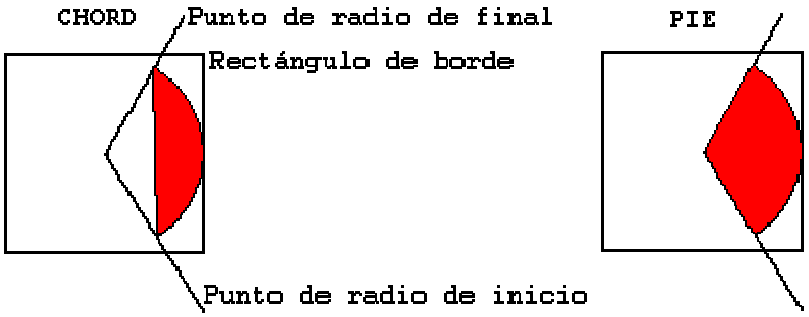
La mayoría de éstas funciones no requieren mayor explicación, los nombres y descripciones dan suficiente información sobre su cometido.

Pintando trozos de elipses, funciones Chord y Pie

Existen dos funciones para trazar figuras rellenas partiendo de una elipse. Podemos clasificar estas figuras en función del número de trazos rectos que contienen.

Según ese criterio, si sólo hay un segmento recto se trata de la figura que se puede trazar con la función Chord, si tiene dos segmentos rectos se trata de la figura que se puede trazar con la función Pie.

Por ejemplo, las formas que se obtienen al partir una galleta son "chords", las que se obtienen al partir una tarta son "pies":

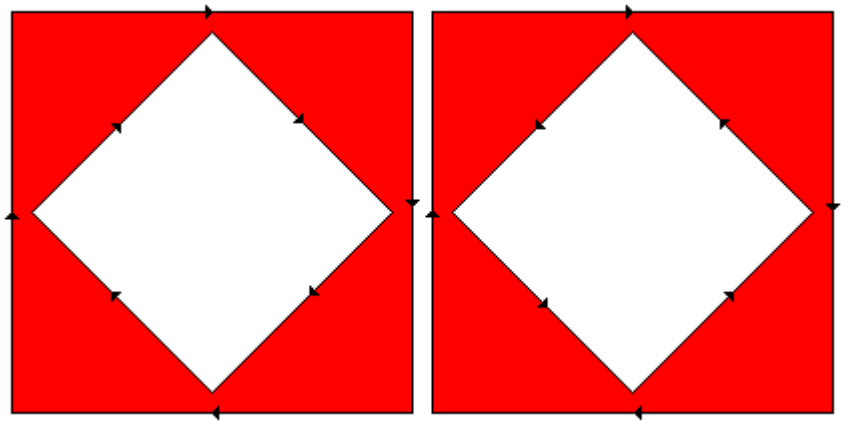


El "centro" de la elipse es el punto medio entre los dos focos, en el caso de la circunferencia, que en realidad es una elipse "degenerada", en la que los dos focos coinciden en un punto, ese punto es el centro.

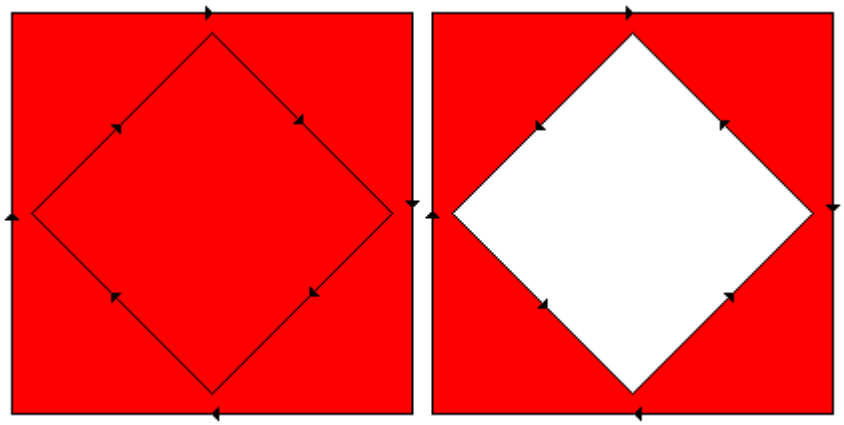
Modos de relleno de polígonos

Existen dos modos de relleno de polígonos. En general la diferencia es mínima, y sólo se manifiesta en casos de polígonos complejos, con varias superposiciones.
Los dos modos de relleno son:

- Alternativo: rellena las áreas entre las líneas impares y pares de cada línea de rastreo. Para ver cómo funciona este modo, imaginemos que recorremos cada línea horizontal de la pantalla de izquierda a derecha. El espacio entre el borde y la primera línea del polígono se deja sin rellenar, el espacio entre la primera línea y la segunda se rellena, el espacio entre la segunda y la tercera, si existe, se deja sin rellenar, etc. Por ejemplo:



- Tortuoso (winding): en este modo se asigna un número a cada región de la pantalla dependiendo del número de veces que se ha usado la pluma para trazar el polígono que la define. Hay que tener en cuenta la dirección en que se recorre cada línea. Las regiones en que ese número no sea nulo, se rellenarán.
Por ejemplo, supongamos la figura siguiente:



Al seguir las líneas del cuadrado externo en el sentido de las flechas, cada uno de los dos cuadrados es rodeado una vez, a cada uno de ellos le asignamos un valor de winding igual a uno.
Al seguir las líneas de cuadrado interno, en el caso de la izquierda recorremos el cuadrado interno en el mismo sentido que la primera vez, por lo tanto, le sumamos a esa figura una unidad a su valor de winding. En el caso de la figura de la derecha, la recorremos en sentido contrario, por lo tanto le restamos una unidad al cuadrado interno, es decir, que en el caso de la derecha, el valor winding del cuadrado interno es cero, y no se rellena.
Si seleccionamos el modo de llenado alternativo, el cuadrado interno no se rellenará nunca, como de hecho sucede en la primera imagen.

Capítulo 22 Objetos básicos del GDI:
La paleta (Palette)

El color es muy importante en Windows, y como todo, es un recurso que tiene sus limitaciones. Cada dispositivo tiene sus capacidades de colores, y el API proporciona funciones para conocer esas capacidades, así como manipularlos, elegirlos, activarlos, etc.
Tal vez, al menos a nivel de pantalla, ya no tenga mucho sentido un capítulo como el presente, las tarjetas gráficas actuales ya no tienen las limitaciones en cuanto a color que tenían hace unos años. Pero en el capítulo dedicado a los mapas de bits veremos que cuando se almacenan en disco o se transmiten, usar paletas nos ahorrará mucho espacio de almacenamiento y mucho tiempo en transmisiones.

Capacidades de Color de los dispositivos

La capacidad de cada dispositivo: pantallas e impresoras, puede variar entre dos y miles o millones de colores. Eso suele ser debido a alguna propiedad física del dispositivo, por ejemplo, una impresora que sólo disponga de un cartucho de tinta negra, sólo podrá visualizar dos colores: blanco y negro. Del mismo modo, una tarjeta gráfica puede estar limitada por la memoria, y sólo disponer de 16 ó 256

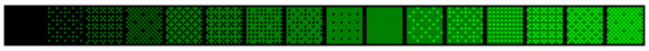
colores, o un monitor en blanco y negro, que sólo disponga de una determinada gama de grises.
A menudo necesitaremos conocer las capacidades de los dispositivos con los que estamos trabajando, y así poder adaptar nuestras aplicaciones de modo que la apariencia de los resultados sea lo más parecido posible a lo que queremos.
Podemos averiguar el número de colores que disponibles en un dispositivo usando la función GetDeviceCaps con el parámetro NUMCOLORS. El número obtenido será el número de colores disponibles para la aplicación.

Definiciones de valores de color

Existen varios modos de codificar los valores de color, Windows usa la síntesis aditiva, expresando las intensidades relativas de los colores primarios: rojo, verde y azul. Para cada color se usan ocho bits, por lo tanto existen 256 valores posibles para cada componente, es decir un máximo de 16.777.216 colores. Los tres bytes forman un triplete RGB y se empaquetan en un entero de 32 bits, de los cuales, los 24 de menor peso se usan para las componentes y el resto a veces se usa para otras funciones que veremos más adelante.
Para almacenar valores de color se usa el tipo COLORREF. Ya hemos usado este tipo antes, por ejemplo para crear plumas con CreatePen o para crear pinceles sólidos con CreateSolidBrush, o en estructuras como LOGPEN. Si se necesita extraer los valores individuales de los componentes de color de un valor COLORREF se pueden usar las macros GetRValue, GetGValue y GetBValue, para rojo, verde y azul respectivamente. También se pueden crear valores de color a partir de los componentes individuales usando la macro RGB.
En las paletas lógicas, se usa la estructura RGBQUAD para definir valores de colores o para examinar valores de componentes.

Aproximaciones de colores y mezclas de pixels (dithering)

En cualquier caso, es posible usar colores sin preocuparse demasiado de las capacidades del dispositivo. Por ejemplo, nada impide crear una pluma verde para una impresora en blanco y negro. Cuando pedimos un color que el dispositivo no admite, Windows escoge otro color entre los que sí puede generar, intentado elegir el más parecido. En nuestro ejemplo, Windows crearía una pluma negra.
El API dispone de la función GetNearestColor que admite como parámetro un valor de color y devuelve el más parecido que puede generar el dispositivo. Esto nos ayuda a predecir qué color obtendremos en cada caso.
Las aproximaciones siempre se usan cuando se eligen colores para plumas o para textos, pero cuando se eligen colores para pinceles sólidos Windows puede intentar simular el color mediante tramas de pixels de distintos colores elegidos entre los que el dispositivo sí puede generar.
En este ejemplo vemos cómo se simulan distintos tonos de verde mezclando los tonos de verde disponibles con negro:



No se dispone de ningún mecanismo para controlar cómo hace Windows estas mezclas, ya que dependen del driver del dispositivo. Lo que sí podemos hacer es crear nuestros propios pinceles usando tramas de mapas de bits.

Mezclas de colores (ROP)

Cuando dibujamos en pantalla usando una pluma o un pincel, no tenemos por qué limitarnos a activar pixels, es posible combinar el nuevo color con el color previo de cada pixel en pantalla. A esto se le denomina mezcla de colores.
Existen distintos modos de mezcla de primer plano, u operaciones binarias de rastreo, que determinan el modo en que se combinan los colores nuevos con los existentes en pantalla previamente. Es posible fundir colores, conservando todos los componentes de ambos colores; enmascarar colores, eliminando o atenuando componentes no comunes; o enmascarar exclusivamente, eliminando o atenuando componentes comunes. Además hay variaciones sobre estas operaciones de mezcla básicas.
Veremos este tema en profundidad más adelante, en un capítulo dedicado a él.
Los colores obtenidos mediante la mezcla también se someten a aproximaciones de color. Si el color obtenido de una mezcla no puede ser mostrado por el dispositivo, Windows genera una aproximación. Como estas operaciones se hacen pixel a pixel, si el color en pantalla proviene de una mezcla de pixels, serán los pixels individuales los que se combinen.
Para seleccionar el modo de mezcla de primer plano se usa la función SetROP2 y para recuperar la actual se usa GetROP2.

Nota: También existe un modo de mezcla de fondo, pero este modo no controla la mezcla de colores. En realidad especifica si el color de fondo será usado cuando se trazan líneas con estilos, pinceles de tramas y texto.

Paletas de colores

Una paleta de colores es un conjunto que contiene valores de colores que pueden ser mostrados en el dispositivo de salida.
En Windows existen dos tipos de paletas: paletas lógicas y paletas de sistema. Dentro de las lógicas existe una especial, la paleta por defecto, que es la que se usa si el usuario no crea una.
Las paletas de colores se suelen usar en dispositivos que, aunque pueden generar muchos colores, sólo pueden mostrar o dibujar con un subconjunto de ellos en un momento dado. Para estos dispositivos, Windows mantiene una paleta de sistema que permite almacenar y manejar los colores actuales del dispositivo.
Windows no permite acceder a la paleta de sistema directamente, en vez de eso, los accesos se hacen mediante una paleta lógica. Además, Windows crea una paleta por defecto para cada contexto de dispositivo. Como programadores, podemos usar los colores de la paleta por defecto o bien crear nuestra propia paleta lógica y asociarla al contexto de dispositivo.
Para determinar si un dispositivo soporta paletas de colores de puede comprobar el bit RC_PALETTE del valor RASTERCAPS devuelto por la función GetDeviceCaps.

La paleta por defecto

Como ya hemos dicho, Windows asocia la paleta por defecto con un contexto cada vez que una aplicación crea un contexto para un dispositivo que soporte paletas de colores. De este modo Windows se asegura de que existen colores disponibles para usar en la aplicación sin necesidad de otras acciones.
La paleta por defecto normalmente tiene 20 entradas, pero ese número depende del dispositivo, y es igual al valor NUMCOLORS devuelto por GetDeviceCaps.
Los colores de la paleta por defecto dependen del dispositivo, en dispositivos de pantalla pueden ser los 16 colores estándar de VGA más cuatro definidos por Windows.
Cuando se usa la paleta por defecto, las aplicaciones pueden usar valores de color para especificar el color de plumas o texto. Si el color requerido no se encuentra en la paleta, Windows aproxima el color usando el más parecido de la paleta. Si una aplicación pide un pincel sólido de un color que no está en la paleta, Windows simula el color mediante mezcla de pixels con colores presentes en la paleta.
Para evitar aproximaciones y mezclas de pixels, se pueden especificar colores para plumas, pinceles y texto usando índices de paleta de colores en lugar de valores de colores. Un índice de paleta de colores es un entero que identifica una entrada en una paleta específica. Se pueden usar índices de paleta en lugar de valores de color, pero se debe usar la macro PALETTEINDEX para crearlos.
Los índices de paleta sólo se pueden usar en dispositivos que soporten paletas de color. Esto puede hacer que nuestros programas sean dependientes del dispositivo, ya que no podremos usar índices para cualquier dispositivo. Para evitarlo, cuando se usa el mismo código para dibujar tanto en dispositivos con o sin paleta, se deben usar valores de color relativos a la paleta para especificar colores de plumas, pinceles o texto. Estos valores serán idénticos a los valores de colores, excepto cuando se crean pinceles sólidos.
En dispositivos con paleta, un color de pincel sólido especificado por un valor de color relativo a la paleta está sujeto a aproximación de color en lugar de a mezcla de pixels. Para crear valores de color relativos a la paleta se debe usar la macro PALETTERGB.
Windows no permite cambiar las entradas de la paleta por defecto. Si se quiere usar otros colores en lugar de los que ésta contiene, se debe crear una paleta lógica propia y seleccionarla en el contexto de dispositivo.

Paleta lógica

Una paleta lógica es una paleta que crea una aplicación y que se asocia con un contexto de dispositivo dado.
Para crear una paleta lógica se usa la función CreatePalette. Antes es necesario llenar la estructura LOGPALETTE, que especifica el número de entradas en la paleta y el valor de cada una, después se debe pasar esa estructura a la función **CreatePalette**. La función devuelve un manipulador de paleta que puede usarse en otras operaciones para identificar la paleta.

```
#define NUMCOLORES 18
...
    PALETTEENTRY Color[NUMCOLORES] = {
       //peRed, peGreen, peBlue, peFlags
       {0,0,0,PC_NOCOLLAPSE},
       {0,20,0,PC_NOCOLLAPSE},
...
       };
    LOGPALETTE *logPaleta;
...
           // Crear una paleta nueva:
           logPaleta                                  =
(LOGPALETTE*)malloc(sizeof(LOGPALETTE) +
              sizeof(PALETTEENTRY) * NUMCOLORES);
           if(!logPaleta)
              MessageBox(hwnd, "No pude bloquear la
memoria de la paleta",
                 "Error", MB_OK);

           logPaleta->palVersion = 0x300;
           logPaleta->palNumEntries = NUMCOLORES;
           for(i = 0; i < NUMCOLORES; i++) {
              logPaleta->palPalEntry[i].peBlue     =
Color[i].peBlue;
              logPaleta->palPalEntry[i].peGreen    =
Color[i].peGreen;
              logPaleta->palPalEntry[i].peRed      =
Color[i].peRed;
              logPaleta->palPalEntry[i].peFlags    =
Color[i].peFlags;
           }

           hPaleta = CreatePalette(logPaleta);
           if(!hPaleta)
              MessageBox(hwnd,  "No  pude  crear  la
paleta", "Error", MB_OK);
           free(logPaleta);
```

Para usar los colores de una paleta lógica, la aplicación selecciona la paleta dentro de un contexto de dispositivo, usando la función SelectPalette. Los colores de la paleta estarán disponibles tan pronto como la paleta sea activada.

```
        case WM_PAINT:
           hDC = BeginPaint(hwnd, &ps);
           hPaletaOld = SelectPalette(hDC, hPaleta,
FALSE);
           hBrush                                    =
CreateSolidBrush(PALETTEINDEX(6));
           hOldBrush   =   (HBRUSH)SelectObject(hDC,
hBrush);
           Rectangle(hDC, 20, 20, 40, 40);
           SelectObject(hDC, hOldBrush);
           DeleteObject(hBrush);
           SelectPalette(hDC, hPaletaOld, FALSE);
           EndPaint(hwnd, &ps);
           break;
```

Es posible limitar el tamaño de las paletas lógicas para permitir las entradas que representen sólo los colores necesarios. Además, no se pueden crear paletas lógicas más grandes que el tamaño máximo de paleta, y ese valor depende del dispositivo. Para obtener el tamaño máximo de la paleta se usa la función GetDeviceCaps para recuperar el valor del valor SIZEPALETTE.

Si bien es posible especificar cualquier color para una entrada de una paleta lógica, es posible que no todos los colores puedan ser generados por el dispositivo. Windows no proporciona una forma de averiguar qué colores son soportados, pero la aplicación puede descubrir el número total de esos colores leyendo la resolución de color del dispositivo. La resolución de color, especificada en bits de colores por pixel, es igual al valor COLORRES revuelto por la función GetDeviceCaps. Un dispositivo que tenga una resolución de color de 18 tiene 262.144 colores posibles. Si una aplicación pide un color no soportado, Windows elige una aproximación apropiada.

Una vez que una paleta lógica a sido creada, se pueden cambiar colores dentro de ella usando la función SetPaletteEntries. Si la paleta lógica ha sido seleccionada y activada, los cambios no afectarán inmediatamente a los colores actualmente mostrados. Para eso es necesario usar las funciones UnrealizeObject y RealizePalette, de ese modo los colores de la pantalla se actualizarán. En algunos casos, puede ser necesario deseleccionar, desactivar, seleccionar y activar una paleta lógica para asegurarse de que los colores se actualizarán exactamente como se requiere. Si se selecciona una paleta lógica en más de un contexto de dispositivo, los cambios en la paleta lógica afectan a todos los contextos de dispositivo en las que ha sido seleccionada.

Se puede cambiar el número de entradas de una paleta lógica usando la función ResizePalette. Si la aplicación reduce el tamaño, las entradas que quedan

permanecen sin cambios. Si se aumenta el tamaño, Windows asigna los colores para cada nueva entrada a negro (0, 0, 0) y el banderín a cero.
También es posible recuperar los valores del color y del banderín para entradas en una paleta lógicas dada mediante la función GetPaletteEntries. Se puede recuperar el índice para una entrada dentro de una paleta lógica que coincida lo más cercanamente posible con un color especificado usando la función GetNearestPaletteIndex.
Cuando no se necesite más una paleta lógica, se debe eliminar usando la función DeleteObject. Hay que asegurarse de que la paleta lógica no permanece seleccionada dentro de un contexto de dispositivo antes de borrarla.

```
        case WM_DESTROY:
           DeleteObject(hPaleta);
           PostQuitMessage(0);
           break;
```

Paleta de sistema

Windows mantiene una paleta de sistema para cada dispositivo que use paletas. Esta paleta contiene los valores de todos los colores que pueden ser mostrados o usados actualmente por el dispositivo. Las aplicaciones no tienen acceso a la paleta de sistema directamente, al contrario, Windows tiene control absoluto de la paleta del sistema y permite el acceso sólo a través de las paletas lógicas.
Para ver el contenido de la paleta del sistema se usa la función GetSystemPaletteEntries. Esta función recupera el contenido de una o más entradas, hasta el número total de entradas en la paleta de sistema. El número total es siempre el mismo que el devuelto para el valor SIZEPALETTE de la función GetDeviceCaps y es el mismo que el tamaño máximo de cualquier paleta lógica dada.
Ya hemos dicho que no es posible modificar los colores de la paleta de sistema directamente, sino sólo cuando se activan paletas lógicas. Antes de activar una paleta, Windows examina cada color requerido e intenta encontrar una entrada en la paleta del sistema que coincida exactamente. Si Windows encuentra ese color, asigna el índice de la paleta lógica para que corresponda con ese índice de la paleta del sistema. Si no lo encuentra, se copia el color requerido en una entrada no usada de la paleta de sistema antes de asignar el índice. Si todas las entradas de la paleta de sistema están en uso, Windows asigna al índice de la paleta lógica la entrada de la paleta de sistema cuyo color sea lo más parecido posible al color requerido. Una vez que los índices han sido asignados, no es posible ignorarlos. Por ejemplo, no es posible usar índices de la paleta de sistema para especificar colores, sólo se permite el uso de índices de la paleta lógica.
Se puede modificar el modo en que los índices serán asignados cuando se seleccionen los valores de la paleta lógica mediante el miembro peFlags de la estructura PALETTEENTRY. Por ejemplo, el banderín PC_NOCOLLAPSE indica a Windows que copie inmediatamente el color requerido en una entrada no usada de la paleta de sistema aunque la paleta de sistema ya contenga ese color.

Además, el flag PC_EXPLICIT indica a Windows que le asigne al índice de paleta lógica un índice esplícito de la paleta de sistema. (Para eso se debe dar el índice de la paleta de sistema en la palabra de menor orden de la estructura PALETTEENTRY).
Las paletas pueden ser activadas como paleta de fondo o como paleta de primer plano espeficando TRUE o FALSE para el parámetro bForceBackground en la función SelectPalette, respectivamente. Sólo puede existir una paleta de primer plano en el sistema al mismo tiempo. Si una ventana o una descendiente suya es la activa actualmente, puede activar una paleta de primer plano. En caso contrario la paleta se activa como paleta de fondo, independientemente del valor del parámetro bForceBackground. La principal propiedad de una paleta de primer plano es que cuando se activa, puede sobrescribir todas las entradas de la paleta de sistema (excepto las estáticas). Windows lo permite marcando las entradas de la paleta de sistema que no sean estáticas como no usadas antes de activar una paleta de primer plano, pudiendo eliminarse todas las entradas usadas. La paleta de primer plano usa todos los colores no estáticos posibles. Las de fondo sólo pueden usar las que permanezcan libres y que estén disponibles para el primero que las solicite. Normalmente, se usan paletas de fondo con ventanas hijas que activen sus propias paletas. Esto ayuda a minimizar el número de cambios en la paleta de sistema.
Una entrada de paleta de sistema no usada es cualquiera que no está reservada y que no contenga un color estático.
En estos tiempos de potentes tarjetas gráficas, con millones de colores y enormes velocidades de proceso, este tipo de preocupaciones ha pasado (felizmente) a la historia. No hace muchos años, muchas tarjetas gráficas limitaban sus paletas a 16 ó 256 colores, aunque fueran capaces de mostrar muchos más. El resultado es que frecuentemente distintas aplicaciones activaban diferentes paletas. Sólo la aplicación que tiene el foco puede activar una paleta de primer plano, de modo que las aplicaciones que perdían el foco también podían ver modificados gran parte de sus colores. El resultado es que los colores de las aplicaciones cambiaban continuamente al cambiar de aplicación, creando un efecto molesto y poco estético.
Las entradas reservadas están marcadas explícitamente con el valor PC_RESERVED. Esas entradas se crean cuando se activan paletas lógicas para animanciones de paleta. Las entradas de color estáticas se crean por Windows y corresponden a los colores de la paleta por defecto. Se puede usar la función GetDeviceCaps para recuperar el valor NUMRESERVED, que especifica el número de entradas de la paleta de sistema reservados para colores estáticos.
Ya que la paleta de sistema tiene un número de entradas limitado, la selección y activación de una paleta lógica para un dispositivo dado puede afectar a los colores asociados con otras paletas lógicas del mismo dispositivo.
Esos cambios de color son especialmente dramáticos cuando se dan en una pantalla. Para asegurarse de que se usan los colores de una paleta lógica seleccionada del modo más fiel hay que resetear la paleta antes de usarla. Esto se hace llamando a las funciones UnrealizeObject y RealizePalette. Usando estas funciones se obliga a Windows a reasignar los colores de la paleta lógica a colores adecuados de la paleta de sistema.

Capítulo 23 Objetos básicos del GDI:
El Mapa de Bits (Bitmap)

Windows usa mapas de bits para muchas cosas. Si te fijas en la ventana de tu aplicación, (ahora probablemente sea un explorador de Internet), verás unos cuantos. Por ejemplo, en la barra de herramientas. Pero también son mapas de bits las flechas de las barras de desplazamiento, los dibujos de los botones de cerrar, maximizar o minimizar, los iconos, etc.

Simplificando, podemos considerar que un mapa de bits es un rectángulo de pixels que en conjunto forman una imagen.

Dentro del API un mapa de bits es una colección de estructuras: una cabecera que contiene información sobre dimensiones, tamaño del array de bits, etc. Una paleta lógica y un array de bits, que relacionan los pixels con la paleta.

Tipos de mapas de bits

Windows trabaja con dos tipos de mapas de bits: dependientes del dispositivo (DDBs) e independientes del dispositivo (DIBs).

Los DDBs provienen de las primeras versiones de Windows, anteriores a la 3.0. Más adelante, debido a ciertos problemas con los dispositivos, se crearon los DIBs.

Pero por el momento esto no nos preocupa mucho, ya lo veremos en profundidad. Lo que nos interesa ahora es cómo usar mapas de bits en nuestros programas, y en eso nos vamos a centrar.

Crear un mapa de bits

Existen funciones para crear mapas de bits, pero están pensadas para generarlos de una forma más o menos dinámica, más que para usar mapas de bits que contengas imágenes ya existentes.

En el presente capítulo nos limitaremos a usar mapas de bits que ya existan en forma de fichero ".bmp". Nuestros programas pueden trabajar con esos ficheros de dos formas: mapas de bits en ficheros de recursos, o mapas de bits almacenados en ficheros en disco.

Fichero de recursos

Esta es la forma de añadir mapas de bits que la aplicación pueda usar como parte de controles, por ejemplo, mapas de bits en menús o botones, pero esos mapas de bits pueden usarse por la aplicación para cualquier otra cosa. Hay que tener en cuenta que esos mapas de bits se incluyen en el ejecutable, por lo tanto no es muy recomendable usar mapas de bits muy grandes, ya que eso aumentará considerablemente el tamaño del fichero ".exe".

El modo de usar estos recursos es añadir una línea BITMAP en nuestro fichero de recursos:

```
#include <windows.h>
#define MASCARA 1000
#define CM_RECURSO 100
#define CM_FICHERO 101

Icono ICON "Food.ico"

Bitmap BITMAP "meninas24.bmp"
MASCARA BITMAP "mascara24.bmp"

Menu MENU
BEGIN
  POPUP "Opciones"
  BEGIN
     MENUITEM "&Bitmap de recurso", CM_RECURSO
     MENUITEM "&Bitmap de fichero", CM_FICHERO
  END
END
```

Esto es una parte del proceso, la otra parte consiste en obtener un manipulador de mapa de bits en nuestra aplicación, para ello se se usa la función LoadBitmap. Esta función requiere dos parámetros, el primero es un manipulador de la instancia que contiene el mapa de bits. Generalmente será la instancia actual, pero los recursos, como veremos en el futuro, también pueden estar contenidos en librerías dinámicas. El segundo parámetro es el identificador del recurso en forma de cadena, o (si hemos usado un entero) el resultado de aplicar la macro MAKEINTRESOURCE a ese entero.
Por supuesto, una vez que no necesitemos el manipulador, debemos eliminarlo usando la función DeleteObject.

```
    static HINSTANCE hInstance;
    static HBITMAP hBitmapRes;
    static HBITMAP hMascara;
...
    switch (msg)                    /* manipulador del
mensaje */
    {
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
```

```
            hBitmapRes     =     LoadBitmap(hInstance,
"Bitmap");
            hMascara       =     LoadBitmap(hInstance,
MAKEINTRESOURCE(MASCARA));
...
        case WM_DESTROY:
            DeleteObject(hBitmapRes);
            DeleteObject(hMascara);
            PostQuitMessage(0);          /*  envía  un
mensaje WM_QUIT a la cola de mensajes */
            break;
```

Fichero BMP

Otro modo de obtener mapas de bits para usarlos en nuestra aplicación es directamente a partir de ficheros BMP, sin necesidad de fichero de recursos. Para leer uno de esos ficheros se usa la función LoadImage.

En realidad, la función LoadImage se puede usar en ambos casos, pero considero que la función LoadBitmap es más cómoda para obtener un manipulador de mapa de bits cuando se trata de recursos.

Necesitamos indicar como manipulador de instancia el valor NULL, si se usa un valor de instancia, generalmente se usará para obtener un mapa de bits de recursos. El segundo parámetro es el nombre del fichero, o el nombre del recurso. El tercer parámetro indica el tipo de imagen, que puede ser un mapa de bits, un icono o un cursor. Los dos siguientes indican un tamaño de imagen, pero no se usan cuando se trata de mapas de bits. El último parámetro permite ajustar algunas opciones, pero sobre todo nos interesa el valor LR_LOADFROMFILE.

```
    static HBITMAP hBitmapRes;
...
    switch (msg)                    /* manipulador del
mensaje */
    {
        case WM_CREATE:
            hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
            hBitmapFil   =   (HBITMAP)LoadImage(NULL,
"Abanicos.bmp", IMAGE_BITMAP,
                0, 0, LR_LOADFROMFILE);
...
        case WM_DESTROY:
            DeleteObject(hBitmapFil);
```

```
            PostQuitMessage(0);         /*  envía  un
mensaje WM_QUIT a la cola de mensajes */
            break;
```

Por supuesto, cuando ya no se necesite, se debe liberar el recurso usando DeleteObject.

Mostrar un mapa de bits

Al contrario que con otros objetos del GDI que hemos manejado hasta ahora, los mapas de bits no se seleccionan directamente en un DC de ventana. Esto se debe a que Windows sólo permite seleccionar los mapas de bits en DC de memoria, y además, sólo en uno al mismo tiempo.

Por lo tanto, para mostrar un mapa de bits tendremos que realizar algunas tareas:

1. Crear un DC de memoria.
2. Seleccionar el mapa de bits en ese DC.
3. Usar una de las funciones disponibles para mostrar el mapa de bits.
4. Borrar el DC de memoria.

El DC de memoria que necesitamos no es un DC cualquiera, debe tratarse de un DC compatible con el dispositivo en el que queramos mostrar el mapa de bits. Para crear uno de esos DCs se usa la función CreateCompatibleDC.

Seleccionar el mapa de bits es una operación similar a seleccionar pinceles, brochas o paletas. Se usa la función SelectObject, el mapa de bits se maneja mediante un manipulador de mapas de bits, un HBITMAP, que habremos obtenido de uno de los modos que hemos explicado más arriba.

En cuanto a las funciones que podemos usar para mostrar mapas de bits, la más sencilla es BitBlt. Pero hay otras, aunque algunas no están disponibles en todas las versiones de Windows.

Finalmente liberamos el DC usando la función DeleteDC.

Funciones de visualización de mapas de bits

Las funciones que veremos de momento son:

BitBlt

Muestra un mapa de bits, sin escalar ni deformar. Se especifica un rectángulo de destino, si el mapa de bits origen es más pequeño se rellena con el color de fondo, si es más grande, ser recorta.

No es necesario mostrar todo el mapa de bits, podemos mostrar sólo un área rectangular del tamaño que queramos, y a partir de la dirección que prefiramos.

Por ejemplo, es muy frecuente crear un único mapa de bits con muchos gráficos reunidos, formando una lista o un mosaico. La función BitBlt nos permite mostrar sólo uno de esos gráficos, sin preocuparnos por el resto:

```
void CTrozo(HDC hDC, HWND hWnd, HBITMAP hBitmap)
```

```
{
    HDC memDC;

    memDC = CreateCompatibleDC(hDC);
    SelectObject(memDC, hBitmap);
    BitBlt(hDC, 135, 225, 40, 40, memDC, 135, 225,
SRCCOPY);
    DeleteDC(memDC);
}
```

Esta función muestra en el rectángulo de la ventana que empieza en las coordenadas (135, 225), y que tiene 40x40 pixels, el trozo del mapa de bits que empieza en las coordenadas (135, 225). Como el mapa de bits se muestra en relación de un pixel por cada pixel del mapa de bits, no habrá distorsión, el trozo de mapa de bits mostrado tendrá el mismo tamaño que el rectángulo en el que se muestra.

StretchBlt

Muestra un mapa de bits o parte de él. El mapa se adapta al rectángulo especificado como destino, estirándose en cada dirección de la forma necesaria para ocuparlo por completo. El modo de estiramiento se puede modificar mediante la función SetStretchBltMode.

Al igual que en el caso anterior, es posible mostrar un trozo del mapa de bits, pero con esta función podemos escalar ese trozo para adaptarlo a la superficie rectangular que queramos:

```
void CAmpliacion(HDC hDC, HWND hWnd, HBITMAP
hBitmap)
{
    HDC memDC;
    BITMAP bm;
    RECT re;

    memDC = CreateCompatibleDC(hDC);
    SelectObject(memDC, hBitmap);
    GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
    GetClientRect(hWnd, &re);
    StretchBlt(hDC, 0, 0, re.right, re.bottom,
memDC,
                    135, 225, 40, 40, SRCCOPY);
    DeleteDC(memDC);
}
```

En este caso se muestra el mismo trozo del mapa de bits que con el ejemplo anterior, pero en lugar de usar el mismo tamaño en la pantalla, el trozo se estirará para adaptarse al área indicada, que no es otra que toda el área de cliente.
Para indicar toda el área de cliente hemos obtenido sus coordenadas mediante la función GetClientRect.
Para obtener el tamaño del mapa de bits usamos la función GetObject que nos devuelve una estructura BITMAP asociada al mapa de bits.

PlgBlt (Sólo en Windows NT)

Muestra el mapa de bits o parte de él, deformándolo para adaptarlo al paralelogramo indicado mediante un array de tres puntos.

Nota: un paralelográmo queda definido por tres puntos, ya que el cuarto se puede calcular a partir de los otros tres. Esto es debido a que los paralelogramos son cuadriláteros con sus lados paralelos dos a dos.

La imagen resultante parece haber sido proyectada sobre una pantalla no paralela al observador.
En lo demás, la función es parecida a StretchBlt. Tammbién requiere que se defina el área del mapa de bits que se mostrará, permitiendo mostrar sólo una parte.
Además, se puede especificar una máscara opcional. La máscara es un mapa de bits monocromo, en la que los pixels de valor uno indican que se debe copiar el pixel, y los de valor cero indican que se debe conservar el fondo.

MaskBlt (Sólo en Windows NT)

Esta es la más complicada de las funciones para mostrar mapas de bits. Funciona de modo análogo a BitBlt, en lo que respecta a que se mantiene la relación de pixels uno a uno, sin deformar la imagen. Pero usa un segundo mapa de bits monocromo como máscara. Esa máscara se puede combinar mediante códigos ROP el mapa de bits con la imagen actual de la ventana y con el pincel actual seleccionado en el contexto de dispositivo.
Para esta función se deben especificar dos mapas de bits. El primero es la imagen a mostrar, el segundo es un mapa de bits monocromo que define una máscara.
Además debemos indicar dos códigos ROP ternarios, uno de los cuales se usa para el fondo y el otro para el primer plano. Si un pixel del mapa de bits de la máscara es un uno, se aplicará el código ROP para el primer plano, si es cero, el del fondo.
Esto nos permite mostrar mapas de bits con formas no rectangulares, con una gran libertad a la hora de relacionar la imagen actualmente en pantalla con la nueva.

Códigos ROP ternarios

Existen quince valores posibles, que agruparemos según su función.
El primer grupo lo componen los códigos ROP en los que el segundo bitmap no tiene importancia ya que no se usa para obtener el resultado final:

- **BLACKNESS:** rellena el área indicada con el color asociado al índice 0 de la paleta física. Este color es negro en la paleta física por defecto.
- **DSTINVERT:** invierte el área de destino. Invertir significa, aplicar el operador de bits NOT para cada valor de color pixel. El resultado en pantalla es un negativo fotográfico.

- **WHITENESS:** Rellena el rectángulo de destino usando el color asociado al índice 1 de la paleta física. (Este color es blanco para la paleta física por defecto.)

Hay dos códigos que implican al pincel asociado actualmente al contexto de dispositivo, que puede ser de cualquier tipo, y no necesariamente un creado a partir de un mapa de bits:

- **PATCOPY:** rellena el área indicada con el pincel actual.
- **PATINVERT:** combina los colores del pincel actual con los colores en el rectángulo destino usando el operador boolenao XOR.

El resto de los códigos ROP implican un segundo mapa de bits:

- **MERGECOPY:** mezcla los colores en el rectángulo de origen con el patrón especificado usando la operación booleana AND.
- **MERGEPAINT:** mezcla los colores invertidos del rectángulo de origen con los colores del rectángulo de destino usando la operación boolena OR.
- **NOTSRCCOPY:** copia el rectángulo de origen invertido al rectángulo de destino.
- **NOTSRCERASE:** combina los colores de los rectángulos de origen y destino usando el operador booleno OR y después invierte el color resultante.
- **PATPAINT:** combina los colores del patrón con los colores invertidos del rectángulo de origen usando el operador booleano OR. El resultado de esta operación se combina con los colores del rectángulo de destino usando el operador booleano OR.
- **SRCAND:** combina los colores de los rectángulos de origen y destino usando el operador booleano AND.
- **SRCCOPY:** copia el rectángulo de origen directamente en el rectángulo de destino.
- **SRCERASE:** combina los colores invertidos el rectángulo de destino con los colores del rectángulo de origen usando el operador booleano AND.
- **SRCINVERT:** combina los colores de los rectángulos de origen y destino usando el operador booleano XOR.
- **SRCPAINT:** combina los colores de los rectángulos de origen y destino usando el operador booleano OR.

Códigos ROP cuádruples

La función MaskBlt requiere como parámetro un código ROP cuádruple, estos códigos se obtienen combinando dos códgos ROP ternarios mediante la macro MAKEROP4.

Pinceles creados a partir de mapas de bits

Es posible crear pinceles de tramas basadas en mapas de bits, como adelantamos en el capítulo 20. Para hacerlo se usa la función CreatePatternBrush.

```
   HBRUSH pincel, anterior;
   HBITMAP hBitmapLazo;

   hBitmapLazo = LoadBitmap(hInstance, "Lazo");
   pincel = CreatePatternBrush(lazo);
   anterior = SelectObject(hDC, pincel);
...
```

```
    SelectObject(hDC, anterior);
    DeleteObject(pincel);
    DeleteObject(hBitmapLazo);
```

Existen además dos funciones para rellenar superficies trabajando con pinceles.

PatBlt

Sirve para rellenar un área rectangular usando el pincel actual. Además, se puede especificar un código ROP para indicar el modo en que deben combinarse el pincel con el fondo. No todos los código ROP se pueden usar con esta función, solo:

ROP	Descripción
PATCOPY	Copia el patrón al mapa de bits de destino.
PATINVERT	Combina el mapa de bits de destino con el patrón usando el operador OR.
DSTINVERT	Invierte el mapa de bits de destino.
BLACKNESS	Pone todas las salidas a ceros.
WHITENESS	Pone todas las salidas a unos.

ExtFloodFill

También sirve para rellenar áreas usando el pincel actual, pero ExtFloodFill permite que el área a rellenar sea irregular. Hay que indicar el punto donde se empieza a rellenar y el color del recinto que delimita el área. No se pueden especificar códigos ROP.

Nota: Existe una función FloodFill sirve para lo mismo que ExtFloodFill, pero proviene de versiones anteriores del API, y actualmente se considera obsoleta, tan sólo se mantiene por compatibilidad.

Estructuras de datos

Existen varias estructuras de datos relacionadas con los mapas de bits, aunque la mayoría están relacionadas con paletas o con el modo de almacenar los datos que contienen o son muy específicos de mapas de bits independientes del dispositivo. Sin embargo hay una estructura que nos resultará muy útil:

BITMAP

La estructura es:

```
typedef struct tagBITMAP {  // bm
   LONG   bmType;
   LONG   bmWidth;
   LONG   bmHeight;
   LONG   bmWidthBytes;
   WORD   bmPlanes;
   WORD   bmBitsPixel;
   LPVOID bmBits;
```

```
} BITMAP;
```

Los datos que más nos interesan son *bmWidth* y *bmHeight*, que indican las dimensiones de anchura y altura, respectivamente, del mapa de bits. El resto de los datos, al menos de momento, no nos interesan.
Para obtener los datos de esta estructura para un mapa de bits concreto, se usa la función GetObject, por ejemplo:

```
    HBITMAP hBitmap;
    BITMAP bm;

    GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
```

El primer parámetro es el manipulador del mapa de bits que nos interesa, el segundo el tamaño de la estructura BITMAP y el tercero un puntero a la estructura que recibirá los datos.

Modos de estiramiento (stretch modes)

Vimos antes, al hablar de la función StretchBlt que existen varios modos diferentes de estiramiento, o mejor dicho, de estrechamiento, ya que estos modos afectan al mapa de bits mostrado cuando se pierden puntos. En concreto existen cuatro, (los otros cuatro son equivalentes para Windows 95), comentaremos algo sobre ellos:

- **BLACKONWHITE:** cuando se pierden puntos, se agrupan realizando una operación booleana AND entre los pixels eliminados. Si el mapa de bits es monocromo, este modo conserva los pixels negros a costa de los blancos.
- **COLORONCOLOR:** los puntos perdidos no se tienen en cuenta. Este modo borra todas las líneas de pixels que no se visualizan, sin intentar preservar su información.
- **HALFTONE:** proyecta los pixels del mapa de bits en el rectángulo de destino, el color medio de los pixels que resultan agrupados se aproxima de este modo al original. Si se activa este modo, hay que usar la función SetBrushOrgEx para cambiar el origen del pincel. Si se falla al hacerlo, habrá un desalineamiento del pincel.
- **WHITEONBLACK:** cuando se pierden puntos, se agrupan realizando una operación booleana OR entre los pixels eliminados. Si el mapa de bits es monocromo, este modo conserva los pixels blancos a costa de los negros.

Los modos BLACKONWHITE y WHITEONBLACK se usan sobre todo con mapas de bits monocromo, los otros dos con mapas de bits en color. El modo HALFTONE proporciona mucho mejor resultado, pero es mucho más lento, además, require usar la función SetBrushOrgEx.
También existen dos funciones relacionadas con estos modos: GetStretchBltMode para averiguar el modo actual de un contexto de dispositivo, y SetStretchBltMode para cambiarlo.

Mapas de bits de stock

Aunque no es probable que nos resulten útiles, también existen mapas de bits de stock, que son los que se usan para las barras de deslizamiento y los botones de minimizar, maximizar, etc.
Podemos obtener esos mapas de bits usando las funciones LoadBitmap y LoadImage, usando NULL como manipulador de instancia y uno de los identificadores especiales para los mapas de bits:

```
    HDC memDC;
    BITMAP bm;
    HBITMAP hBitmap;
    RECT re;

    memDC = CreateCompatibleDC(hDC);
    hBitmap            =            LoadBitmap(NULL,
MAKEINTRESOURCE(OBM_CLOSE));
    SelectObject(memDC, hBitmap);
    GetObject(hBitmap, sizeof(BITMAP), (LPSTR)&bm);
    BitBlt(hDC,  10,  10,  bm.bmWidth,  bm.bmHeight,
memDC, 0, 0, SRCCOPY);
    DeleteObject(hBitmap);
    DeleteDC(memDC);
```

Capítulo 24 Objetos básicos del GDI: La Fuente (Font)

Llegamos por fin a un objeto básico que se seguro que necesitamos en casi todos nuestros programas, y que estarías echando de menos: el texto.
Veremos en este capítulo que si bien, mostrar texto en pantalla es fácil, (basta con una función del API), el tema no es tan sencillo como pudiera pensarse, ya que Windows nos ofrece muchas posibilidades a la hora de trabajar con texto. Podemos elegir la forma, el tamaño, orientación, estilo, espaciado... de cada fuente. En este capítulo aprenderemos a mostrar texto y también a personalizarlo.

Mostrar un texto simple

El API siempre dispone de objetos de stock, de modo que si sólo queremos mostrar un texto en nuestra ventana, podemos usar la fuente por defecto para ello, sin complicarnos la vida.
La función más simple para mostrar texto es TextOut, que usa la fuente activa.
La forma de usarla es tan sencilla como indicar un manipulador de contexto de dispositivo, las coordenadas de inicio del texto, el propio texto, y el número de

caracteres a mostrar. Por supuesto, seguiremos usando el mensaje WM_PAINT para actualizar la pantalla:

```
        case WM_PAINT:
           hDC = BeginPaint(hwnd, &ps);
           TextOut(hDC, 10, 10, "Hola, mundo!", 12);
           TextOut(hDC,  20,  30,  "Curso  WinAPI  con
Clase.", 23);
           EndPaint(hwnd, &ps);
           break;
```

Cambiar el color del texto

El texto mostrado por el ejemplo anterior usa la fuente del sistema, con el color por defecto: letras negras sobre fondo blanco.
Esto no resulta muy elegante, que digamos. Es bastante rudimentario y, desde luego, no es lo mejor que sabremos hacer.
Podemos cambiar el color del fondo, usando la función que ya conocemos SetBkColor, o podemos evitar el parche de color correspondiente al fondo, de modo que las letras se muestren sobre el contenido actual del fondo, sea del color que sea. Para lograr esto bastará con activar el modo transparente para el fondo, con la función que ya hemos usado: SetBkMode.

```
           SetBkMode(hDC, TRANSPARENT);
```

El siguiente paso es modificar el color del texto, para ello disponemos de otra función del API: SetTextColor, que precisa un manipulador de contexto de dispositivo y una referencia de color, para indicar el nuevo color del texto.

```
           SetTextColor(hDC, RGB(255,0,0));
```

Si necesitásemos averiguar el color actual del texto, podemos usar la función GetTextColor.

Crear fuentes personalizadas

El siguiente paso es aprender a usar cualquiera de las fuentes disponibles en el sistema, variando tanto el tipo, como el tamaño, orientación y estilo. Los parámetros que podemos cmabiar en una fuente son muchos, de modo que intentaremos explicar cada uno de ellos para que nos sea más sencillo adaptar las fuentes a nuestros gustos o necesidades.
Para ello disponemos de dos funciones: CreateFont y CreateFontIndirect. La primera precisa que le demos 14 parámetros que definen la fuente, la segunda crea la fuente a partir de los mismos parámetros, almacenados en una estructura LOGFONT.

Altura y anchura media de carácter

Existen algunos parámetros clave a la hora de medir una fuente, veamos algunos de ellos:

Línea de base es la línea sobre la que se apoyan las letras, en el caso de letras que sobresalen por debajo, la parte que apoya sobre esta línea es, generalmente, la del óvalo de la letra. La línea de base es lo que nos indica la inclinación de un texto.

Punto es la unidad de medida para fuentes. Un punto es, aproximadamente, 1/72 de pulgada, es decir, una fuente de 72 puntos de altura tendrá una pulgada de alto.

Celda es un rectángulo imaginario que contiene un carácter completo.

El tamaño de una fuente se define por dos parámetros: altura y anchura. Esto es evidente, ya que el texto tiene dos dimensiones. Sin embargo, aunque la altura es un parámetro claro, la anchura no lo es tanto. En algunas fuentes, los caracteres no tienen un ancho constante, sino proporcional a su forma, es decir, letras como la 'i' son más estrechas que otras, como la 'm'.

Estas fuentes, en la que cada carácter tiene una anchura diferente, son conocidas como de anchura proporcional; por contra, el otro tipo, en las que todos los caracteres tienen la misma anchura, se conoce como fuentes de anchura no proporcional:

Fuente no proporcional

Fuente proporcional

Es por eso que cuando hablamos de anchura de fuentes nos referimos a anchura media de carácter.

Los valores de altura son igualmente imprecisos, ya que exiten caracteres de distintas alturas, por ejemplo, las letras como 'p', 'q' y 'g' sobresalen por la parte inferior, del mismo modo que las mayúsculas, o letras como la 't', 'l' y 'f' sobresalen por arriba. Normalmente se determina la altura como la distancia entre la parta inferior de la letra 'g' y la superior de la 'M'.

Es frecuente referirse también a la altura de la celda.

Cuando creamos fuentes, los valores de altura pueden ser positivos, negativos o cero. Los valores negativos indican medidas en función de la altura del carácter, y los positivos en función de la altura de la celda.

El valor nulo indica que se tome el valor por defecto para la altura.

En el caso de la anchura, el valor puede ser positivo o nulo. El valor positivo será el que se tome como anchura media, si es cero, se tomará el valor por defecto, que dependerá en cada caso del valor de altura elegido.

El ángulo de escape
Cada carácter puede tener un ángulo sobre la línea de base. Podemos, por ejemplo, inclinar los caracteres 90º y escribir una línea horizontal:

El ángulo de orientación
Se refiere al ángulo formado por la línea de base con el eje x del dispositivo. Cuando creamos fuentes tenemos una precisión de décimas de grados para precisar dicho ángulo, de modo que un valor de 900 indica 90º.

Peso
El peso indica cómo de gruesos son los trazos que se usan para mostrar el carácter, es lo que diferecia un carácter en negrita de uno normal. Tenemos mil valores posibles para el peso, los valores más bajos indican trazos finos, los más altos, trazos gruesos.
Como ayuda existen ciertas constantes predefinidas para asignar a este parámetro.

Cursiva
Se trata de un banderín, si se activa (valor TRUE), se generará una fuente cursiva, si se desactiva, una fuente vertical.

Subrayado
Otro banderín, si se activa se generará una fuente subrayada, en caso contrario, una normal.

Tachado
Un tercer banderín, que si se activa genera una fuente tachada.

Conjunto de caracteres
Permite elegir entre distintos conjuntos de caracteres. Recordemos que es ASCII no es el único conjunto existente, existen otros, y el API nos permite seleccionar algunos de ellos. Entre los más corrientes están, por ejemplo: el Windows, el Unicode^{tm} y el de símbolos.

Precisión de salida
Básicamente, nos permite elegir una fuente de dispositivo, matricial o TrueType, si existen varias de distinto tipo y el mismo nombre.

Precisión de recorte
Afecta al tipo de rotación de caracteres cuando cambiamos la orientación de la fuente.

Calidad
Permite seleccionar, a la hora de mostrarla en pantalla, la calidad de la fuente.

Paso y familia
Sirve para indicar un tipo alternativo de fuente cuando la seleccionada no está disponible o no se especifica una fuente concreta.

Nombre
Nombre de la fuente seleccionada. Los ficheros de fuentes almacenan el aspecto visual de cada fuente. Esto nos da una enorme posibilidad a la hora de mostrar textos o símbolos.

Fuentes de stock

AL igual que con otros objetos del GDI que hemos usado antes, en el caso de las fuentes también disponemos de seis fuentes de stock, que podemos seleccionar mediante la función GetStockObject.
En concreto se trata de las siguientes:

Valor	Significado
ANSI_FIXED_FONT	Especifica una fuente de espacio no proporcional basada en el conjunto de caracteres de Windows. Normalmente se usa una fuente Courier.
ANSI_VAR_FONT	Especifica una fuente de espacio proporcional basada en el conjunto de caracteres de Windows. Normalmente se usa una fuente MS Sans Serif.
DEVICE_DEFAULT_FONT	Especifica la fuente por defecto para el dispositivo especificado. Se trata, típicamente, de la fuente System para dispositivos de visualización. Para algunas impresoras matriciales esta fuente es una que reside en la propia impresora. (Imprimir con esa fuente suele ser mucho más rápido que hacerlo con otra.).
OEM_FIXED_FONT	Especifica una fuente de espacio no proporcional basada en un conjunto de caracteres OEM. Para ordenadores IBM® y compatibles, la fuente OEM está absada en el conjunto de caracteres del IBM PC.
SYSTEM_FONT	Especifica la fuente System. Es una fuente de espacio proporcional basada en el conjunto de caracteres Windows, y se usa por el sistema operativo para mostrar los títulos de las ventanas, los nombres de menú y el texto en los cuadros de diálogo. La fuente System siempre está disponible. Otras fuentes sólo están disponibles si han sido instaladas.
SYSTEM_FIXED_FONT	Especifica una fuente de espacio no proporcional compatible con la fuente System en versiones de Windows anteriores a la 3.0.

Alineamientos de texto

Cuando mostramos un texto usando la función TextOut, ExtTextOut (que aún no hemos visto), indicamos unas coordenadas para situar el texto en el dispositivo. Estas coordenadas pueden referirse a diversos puntos dentro del texto. Podemos referirnos a la línea de base, al centro del texto, a la esquina superior izquierda, a la esquina superior derecha, etc.

En concreto, tenemos las siguientes opciones:

- TA_BASELINE: El punto de referencia es la línea de base del texto.
- TA_BOTTOM: El punto de referencia es el borde inferior del rectángulo externo que contiene el texto.
- TA_TOP: El punto de referencia es el borde superior del rectángulo que contiene el texto.
- TA_CENTER: El punto de referencia se alinea horizontalmente con el centro del rectángulo que contiene el texto.
- TA_LEFT: El punto de referencia es el borde izquierdo del rectángulo que contiene el texto.
- TA_RIGHT: El punto de referencia es el borde derecho del rectángulo que contiene el texto.
- TA_NOUPDATECP: El valor del cursor no se actualiza después de mostrar el texto.
- TA_UPDATECP: El valor del cursor se actualiza después de mostrar el texto.

Pero no sólo eso, estas opciones se pueden combinar, aunque no de cualquier modo. Sólo se pueden agrupar valores tomando uno o ninguno de cada uno de los grupos:

- TA_LEFT, TA_RIGHT y TA_CENTER
- TA_BOTTOM, TA_TOP y TA_BASELINE
- TA_NOUPDATECP y TA_UPDATECP

Podemos, por ejemplo, combinar el valor TA_LEFT con TA_BASELINE o TA_CENTER con TA_TOP, etc. Para combinarlos se usa el operador de bits OR (|).

Para cambiar la alineación del texto se usa la función SetTextAlign, para obtener el valor actual se usa GetTextAlign.

Pero averiguar si uno de los bits está activo en el valor actual de alineamiento no es tan sencillo como pudiera parecer a primera vista. Los valores TA_xxx son bits dentro de un valor de alineamiento, y como hemos visto esos bits pueden estar combinados entre ellos.

Si queremos comprobar si el valor de alineamiento actual contiene el valor TA_LEFT no bastará con comparar el valor de retorno de GetTextAlign con TA_LEFT, ya que el valor actual puede tener también los valores TA_BOTTOM, TA_TOP, TA_BASELINE, TA_NOUPDATECP o TA_UPDATECP. Tampoco bastará con un AND de bits, ya que no sabemos si, por ejemplo, TA_CENTER equivale a TA_LEFT |TA_RIGHT. (Podría ser).

Los pasos a seguir son:

1. Aplicar el operador de bits OR a los bits del grupo al que pertenece el valor que queremos verificar.

2. Aplicar el operador de bits AND entre el resultado anterior y al valor retornado por GetTextAlign.
3. Verificar la igualdad entre ese resultado y la bandera a verificar.

En nuestro ejemplo, tenemos:

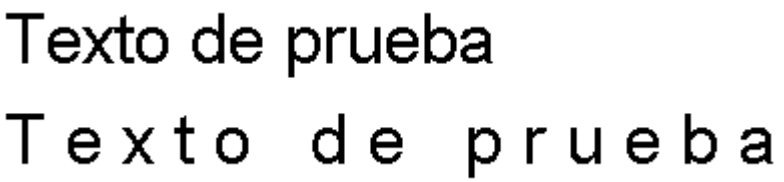
1. x = TA_LEFT | TA_RIGHT | TA_CENTER
2. y = x & GetTextAlign(hDC);
3. Verificar si y == TA_LEFT

Si la alineación actual es, por ejemplo, TA_RIGHT | TA_BASELINE, y queremos comprobar si contiene el valor TA_LEFT, la sentencia C puede ser como esta:

```
if(TA_LEFT == ((TA_LEFT | TA_RIGHT | TA_CENTER) & GetTextAlign(hDC)))...
```

Separación de caracteres

Normalmente, cuando mostramos texto en un dispositivo, la separación entre caracteres está predeterminada, y depende del diseño de la fuente. Pero, con el fin de hacer que el texto sea más ancho, podemos aumentar la separación entre caracteres, usando la función SetTextCharacterExtra.



Del mismo modo, podemos comprimir el texto usando valores de separación negativos.

Por último, podemos recuperar el valor de separación para un contexto de dispositivo determinado usando la función GetTextCharacterExtra.

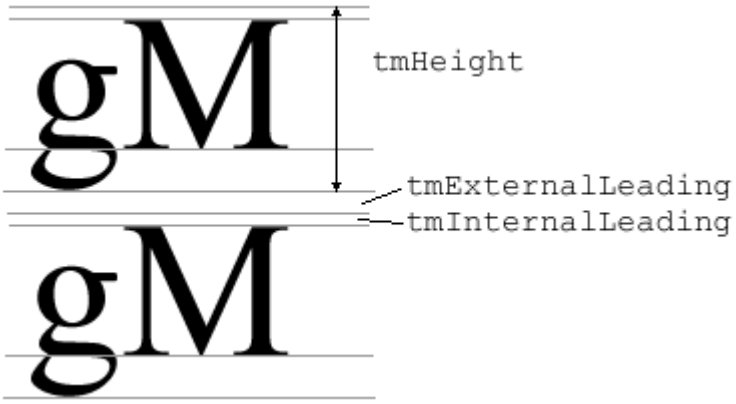
Medidas de cadenas

A menudo nos interesa conocer las medidas que van a tener las cadenas en el dispositivo antes de mostrarlas, por ejemplo, para situarlas correctamente, separar las líneas adecuadamente, formatear párrafos, etc.

Disponemos de varias funciones en el API para esta tarea.

Empezaremos por la función GetTextMetrics, que nos proporciona datos sobre la fuente actual de un contexto de dispositivo en una estructura TEXTMETRIC.

Esta estructura nos informa principalmente sobre las medidas verticales de las líneas de texto, y nos da alguna información sobre medidas horizontales, aunque en este caso, algo menos precisas.



Justificar texto

El API también nos proporciona funciones para mostrar texto justificado: que se ajusta a los márgenes derecho e izquierdo del área de visualización.
Para ello deberemos usar dos funciones de forma conjunta. Por una parte GetTextExtentPoint32, que nos proporciona información sobre el tamaño de una línea de texto. Por otra, la función SetTextJustification, que prepara las cosas para que la siguiente función de salida de texto TextOut incluya la separación apropiada entre palabras.
La función GetTextExtentPoint32 sirve para calcular el espacio necesario de pantalla necesario para mostrar una cadena. Esta información la podemos usar, por una parte, para averiguar si cierta cadena cabe en el espacio en que queremos mostrarla, y por otra, para saber cuanto espacio sobra, si es que cabe.
El espacio sobrante se debe repartir entre las palabras de la cadena a justificar, eso se hace mediante la función SetTextJustification, que necesita como parámetros el manipulador del DC, el espacio extra, y el número de espacios entre palabras que contiene la cadena.
Hay que usar la función SetTextJustification con cuidado, ya que sus efectos permanecen hasta la siguiente llamada. Es decir, si usamos esta función para justificar una línea, y no anulamos su efecto, subsiguientes llamadas a GetTextExtentPoint32 podrán falsear la medida de la cadena, ya que se usará más espacio entre palabras para calcular la longitud de la cadena.
Para anular el efecto de la función se usa la misma función, pero con un valor nulo en el segundo parámetro, y también en el tercero, aunque en realidad ese parámetro se ignora si el segundo es nulo.

```
   char *cadena = "Para ello deberemos usar dos
funciones de";
   SIZE tam;
   RECT re;

   GetClientRect(hwnd, &re);   // Obtener tamaño de
la ventana
```

```
   SetTextJustification(hDC,   0,   0);   //   Anula
cualquier justificación previa
   GetTextExtentPoint32(hDC, cadena, strlen(cadena),
&tam); // Calcular tamaño de cadena
   SetTextJustification(hDC,     re.right-tam.cx-20,
espacios[i]); // Justificar
   TextOut(hDC, 10, 10, cadena, strlen(cadena));  //
Mostrar cadena
```

Capítulo 25 Objetos básicos del GDI: Rectángulos y Regiones

Rectángulos

Los rectángulos se usan en Windows para muchas cosas, y disponen de estructuras y funciones dedicadas a manejarlos.
Ya los hemos usado, por ejemplo, para invalidar parte de la ventana y obligar al Windows a actualizarla, para ello usamos la función InvalidateRect, también los hemos usado para obtener las dimensiones del área de cliente con GetClientRect.
Lo primero que tenemos que ver es la estructura RECT, aunque ya la hemos usado muchas veces en este curso.
Esta estructura define un rectángulo mediante las coordenadas de la parte izquierda, superior, derecha e inferior del rectángulo: left, top, right y bottom. Podemos decir que estos valores definen dos puntos: la esquina superior izquierda y la inferior derecha. Es importante tener claro esto, ya que si la coordenada izquieda es mayor que la derecha, o la superior mayor que la inferior, no estaremos definiendo un rectángulo, al menos desde el punto de vista del API.

Funciones para trabajar con rectángulos

Veremos a continuación algunas de las funciones para trabajar con rectángulos.

Asignar rectángulos

En cuanto a las funciones de asignación, tenemos:

Función	Utilidad
SetRect	Sirve para asignar valores a un rectángulo.
CopyRect	Hace una copia de un rectángulo.
SetRectEmpty	Crea un rectángulo vacío.

Un rectángulo vacío es aquel en el que todas sus coordenadas son cero.

Comparaciones de rectángulos

Tenemos tres funciones para comparar rectángulos:

Función	Utilidad
IsRectEmpty	Nos dice si un rectángulo es vacío.
EqualRect	Nos dice si dos rectángulos son iguales.
PtInRect	Nos dice si un punto está dentro de un rectángulo. Un punto en el lado inferior o en el derecho se considera fuera del rectángulo.

Modificar rectángulos

Para modificar rectángulos, tenemos otras dos:

Función	Utilidad
InflateRect	Nos permite agrandar o reducir un rectángulo.
OffsetRect	Nos permite trasladar un rectángulo.

Operaciones con rectángulos

Por último, para operar con rectángulos, tenemos otras tres:

Función	Utilidad
IntersectRect	Obtiene la intersección de dos rectángulos.
UnionRect	Obtiene la unión de dos rectángulos.
SubtractRect	Obtiene la diferencia emtre dos rectángulos.

La intersección de dos rectángulos es siempre un rectángulo, hay un caso especial, cuando los rectángulos no tienen ninguna superficie en común. En ese caso, el resultado es un rectángulo vacío.

En el caso de uniones, se define la unión de dos rectángulos como el rectángulo más pequeño que incluye ambos rectángulos.

La función de diferencia es algo más restrictiva con los rectángulos que admite como parámetros, ya que deben cortarse completamente al menos en uno de los ejes.

Veremos el uso de rectángulos sobre la marcha, como los hemos visto hasta ahora, sin haberlos explicado en detalle.

Regiones

Las regiones son figuras como rectángulos, elipses o polígonos, o combinaciones de ellos. Y sus aplicaciones son análogas a las de los rectángulos, aunque como veremos, más completas.

Las regiones se manejan mediante manipuladores, y se puede usar para invalidar partes del área de cliente, InvalidateRgn.

Funciones para regiones

Veremos ahora algunas de las funciones de las que disponemos para trabajar con regiones.

Crear regiones

Como en el caso de los rectángulos, disponemos de varias funciones para crear regiones:

Función	Utilidad
CreateRectRgn	Crea una región rectángular.
CreateRectRgnIndirect	Crea una región rectángular a partir de una estructura RECT.
CreateRoundRectRgn	Crear una región rectángulas con las esquinas redondeadas.
CreateEllipticRgn	Crea una región elíptica.
CreateEllipticRgnIndirect	Crea una región elíptica a partir de una estructura RECT que define el rectángulo que inscribe a la elipse.
CreatePolygonRgn	Crea una región poligonal.
CreatePolyPolygonRgn	Crea una región compuesta por varios polígonos.

Cualquier región se puede seleccionar para un contexto de dispositivo, usando la función SelectObject.

Disponemos de un amplio conjunto de operaciones que se pueden realizar con regiones: combinarlas, compararlas, obtener sus dimensiones, pintarlas, recuadrarlas, etc.

Combinar regiones

Para combinar regiones se usa la función CombineRgn, esta función permite combinar dos regiones de cinco formas diferentes:

Combinación	Resultado
RGN_AND	La nueva región es la intersección de las dos regiones combinadas.
RGN_COPY	El resultado no tiene en cuenta la segunda región, es siempre una copia de la primera.
RGN_DIFF	La nueva región es la diferencia entre las dos regiones, la primera menos la parte común.
RGN_OR	La nueva región es la suma de las dos regiones combinadas.
RGN_XOR	La nueva región es la suma de las dos regiones combinadas, excluyendo las partes comunes.

Comparar regiones

Podemos comparar dos regiones usando la función EqualRgn, dos regiones se consideran iguales si tienen el mismo tamaño y forma.

Rellenar regiones

Para rellenar regiones se usa la función FillRgn, al igual que vimos en el capítulo 21, el modo de relleno de polígonos también afecta a la hora de rellenar regiones.

Podemos por lo tanto, usar las funcones SetPolyFillMode y GetPolyFillMode para cambiar o consultar el modo de relleno actual.
Otra función parecida es PaintRgn, que usa el pincel actual, en lugar de tener que especificar uno. Los modos de relleno de polígonos afectan del mismo modo que en FillRgn.
La función InvertRgn invierte los colores presentes en la pantalla para la región especificada. En blanco y negro, invertir significa cambiar los pixels blancos por negros y viceversa. En color, el resultado depende del tipo de dispositivo.
El otro modo de mostrar una región es enmarcarla, para eso se usa la función FrameRgn. Enmarcar una región significa trazar una línea a su alrededor, para lo que hay que especificar un pincel y la anchura y altura del marco.

Mover una región

Mediante la función OffsetRgn podemos desplazar una región en cualquier dirección.

Comprobar posiciones

Tenemos dos funciones para hacer verificaciones sobre regiones, PtInRegion verifica si un punto está en el interior de una región o no. Esto nos será útil cuando trabajemos con el ratón, para saber si el cursor pasa por determinadas zonas de la ventana.
La otra función es RectInRegion, que verifica si alguna parte de un rectángulo está dentro de una región determinada.
En próximos capítulos veremos como podemos usar una región o un camino para definir un área de recorte. Fuera de la zona definida como área de recorte no se producirá ninguna salida gráfica. Esto nos permite crear formas elaboradas.

Destruir regiones

Como otros objetos del GDI, hay que destruir las regiones cuando ya no nos hagan falta, liberando de ese modo los recursos usados. Para destruir una región se usa la función DeleteObject.

Capítulo 26 Objetos básicos del GDI: El camino (Path)

Se usan para crear figuras complejas, a base de unir segmentos rectos con curvas y líneas Bézier. Estas figuras también pueden contener zonas rellenas y texto.
Los caminos siempre están asociados a un contexto de dispositivo, pero al contrario que otros objetos del GDI, no existe un camino por defecto.

Crear un camino

Para crear un camino hay que definir los puntos que lo componen, esto se hace usando funciones de trazado del GDI entre las llamadas a las funciones BeginPath y EndPath.
Las funciones que se pueden usar dentro de un camino son:

AngleArc	LineTo	Polyline

Arc	MoveToEx	PolylineTo
ArcTo	Pie	PolyPolygon
Chord	PolyBezier	PolyPolyline
CloseFigure	PolyBezierTo	Rectangle
Ellipse	PolyDraw	RoundRect
ExtTextOut	Polygon	TextOut

```
                    BeginPath(hdc);
                    SetBkMode(hdc, TRANSPARENT);
                    TextOut(hdc,  10,10,  "Con  Clase",
9);
                    Rectangle(hdc, 0,0,10,10);
                    EndPath(hdc);
```

La función CloseFigure sirve para cerrar figuras irregulares creadas a partir de segmentos rectos y/o curvas.
En cualquier momento, antes de cerrar un camino, podemos eliminarlo usando AbortPath.

Operaciones con caminos

En el momento de cerrar el camino, llamando a EndPath, se selecciona el camino y se borra el previamente seleccionado para ese DC. A partir de ese momento tenemos varias opciones:

StrokePath	Trazar la línea definida por el camino, usando la pluma actual.
FillPath	Pintar el interior del camino, usando el pincel actual.
StrokeAndFillPath	Ambas cosas.
PathToRegion	Convertir el camino en una región.
FlattenPath	Vectorizar el camino, convertir las curvas sen series de segmentos rectos que se aproximen.
GetPath	Recuperar las coordenadas y tipos de los puntos que componen el camino.
SelectClipPath	Convertir el camino en un camino de recorte.

Como en anteriores ocasiones, el proceso de rellenar figuras está sujeto al modo de relleno de polígonos, podemos obtener ese modo llamando a GetPolyFillMode y cambiarlo usando SetPolyFillMode.
El tema de recortes se trata con detalle en el siguiente capítulo.

Capítulo 27 Objetos básicos del GDI: El recorte (Clipping)

El recorte nos permite limitar las salidas del GDI a una determinada zona, definida por una región o por un camino.
La región de recorte es uno de los objetos del GDI que podemos seleccionar en un contexto de dispositivo. Del mismo modo que seleccionamos pinceles, brochas, fuentes, etc.

Regiones de recorte y el mensaje WM_PAINT

Algunos contextos de dispositivo tienen una región de recorte por defecto, por ejemplo, si obtenemos un DC mediante la función BeginPaint, el DC tendrá una región de recorte correspondiente a la región invalidada que ha provocado el mensaje WM_PAINT.
Ya hemos usado la función InvalidateRect para forzar la actualización de parte del área de cliente, normalmente usamos un rectángulo nulo, con lo que se actualiza toda la ventana.
En ese caso, la región de recorte del DC será el rectángulo especificado.
También podemos usar la función InvalidateRgn. Tanto en un caso como en el otro, las sucesivas llamadas a estas funciones actualizan la región de recorte, de modo que la función BeginPaint obtiene la región de recorte resultante.
Otras formas en que esa región se actualiza es cuando parte del área de cliente se oculta proque otras ventanas se superponen. Los rectángulos de esas ventanas componen la región de recorte final, que se recibe al procesar el mensaje WM_PAINT.
Si se obtiene un DC mediante CreateDC o GetDC, no tendremos una región de recorte por defecto.

Funciones relacionadas con el recorte

Existen algunas funciones que se pueden aplicar a la región de recorte asociada a un DC sin necesidad de tener un manipulador de región, directamente sobre el DC:

PtVisible	Nos sirve para determinar si un punto está dentro de los límites de la región asociada a un DC.
RectVisible	Nos permite determinar si una parte de un rectángulo será visible en en DC.
OffsetClipRgn	Para mover la región de recorte en el desplazamiento que queramos.
ExcludeClipRect	Elimina un área rectángular de la región de recorte

	actual.
IntersectClipRect	Limita la región de recorte actual a su intersección con un rectángulo dado.

Seleccionar regiones de recorte

Además de las funciones que hemos visto: InvalidateRect e InvalidateRgn, podemos crear una región de recorte directamente a partir de una región.
Como vimos en el capítulo 25, existen funciones que se aplican directamente sobre regiones. Podemos aplicar estas funciones a la región de recorte si previamente obtenemos esa región mediante GetClipRgn, y una vez modificada, volvemos a aplicar la región de recorte mediante SelectClipRgn.
Si usamos un manipulador de región NULL, crearemos una región de recorte nula.
En el ejemplo de este capítulo, cada vez que se debe dibujar el área de cliente, el contenido es diferente, (se trata de líneas paralelas, cada vez más juntas, y alternativamente verticales y horizontales, y en colores rojo y azul). De este modo podemos ver claramente la región de recorte, ya que únicamente esa región se trazará cada vez.
Hemos puesto una opción para crear una zona de recorte rectángular, que obtenermos usando la función InvalidateRect, y otra para una zona elíptica, que obtenemos usando InvalidateRgn con una región que hemos creado mediante CreateEllipticRgn.
Un detalle importante es que la región de recorte se va construyendo a medida que se invalidan las distintas zonas, y sólo cuando el sistema "decide" procesar el mensaje WM_PAINT se actualiza la región de recorte tal como existe en ese momento.
La decisión de procesar el mensaje WM_PAINT depende de la carga de trabajo del sistema, y del modo en que se procesan las distintas órdenes de invalidar, de modo que siempre tiene algo de aleatorio.
También podemos experimentar lo que pasa si partes de la ventana de cliente se tapan por otras ventanas. Veremos que se crea una región con todas las zonas del área de cliente que se han tapado, y que al pasar a primer plano, sólo esas zonas se actualizan.

Caminos de recorte

Además de regiones, podemos usar caminos para definir áreas de recorte. Los caminos nos proporcionan algo más de flexibilidad, ya que podemos usar curvas Bézier para definir sus figuras.
Normalmente usaremos caminos para definir áreas de recorte con el objetivo de crear efectos especiales, como hicimos en el ejemplo del capítulo anterior, creando un camino a partir de un texto, y dibujando puntos aleatoriamente dentro de ese camino.
En el ejemplo de este capítulo usamos un camino de recorte, y también una región para crear efectos con puntos o líneas que se adaptan a ciertas figuras.
Se usa la función SelectClipPath para crear un área de recorte a partir de un camino, y la función SelectClipRgn para hacerlo a partir de una región.

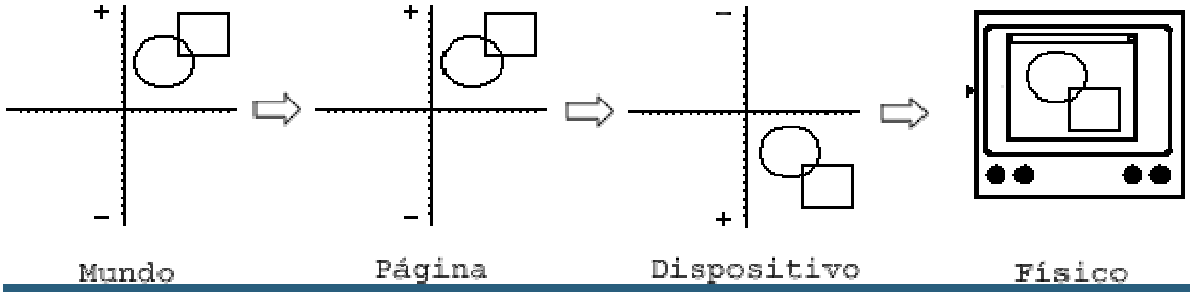
Capítulo 28 Objetos básicos del GDI: Espacios de coordenadas y transformaciones

Definiciones

Un **sistema de coordenadas** es una representación del espacio plano basado en un sistema Cartesiano. Es decir, dos ejes perpendiculares.
En Windows, el espacio es limitado, es decir, el valor máximo de las coordenadas está acotado.
En Windows se usan cuatro sistemas de coordenadas:

- El del mundo. Se refiere al espacio de la aplicación. Permite usar 2^{32} unidades en cada eje.
- El de la página. Es lo que normalmente denominamos espacio lógico, es decir, el espacio donde se usan las unidades lógicas, aunque también en el espacio del mundo trabajaremos con unidades lógicas. Permite usar 2^{32} unidades en cada eje.
- El del dispositivo. En este espacio podemos trabajar con unidades como pulgadas o milímetros. Permite usar 2^{27} unidades en cada eje.
- El del dispositivo físico. Generalmente se refiere al área de cliente, aunque también puede tratarse de la página de la impresora o del ploter. Su tamaño es variable, y depende de cada dispositivo.

Las **transformaciones** son algoritmos que se usan para hacer conversiones entre el espacio de coordenadas del mundo y el de la página, de modo que nos es posible realizar cambios de escala, rotaciones, traslaciones, deformaciones o reflexiones.
Esta característica se introdujo en el API de Win32, y no está disponible para versiones de Windows 95 y anteriores.
En Windows usaremos también otro tipos de proyecciones de coordenadas: el mapeo. El **mapeo** consiste en una transformación, aunque algo más limitada, ya que sólo permite escalar, trasladar y cambiar la orientación de los ejes.
El mapeo se aplica entre el espacio de página y el de dispositivo, y existe desde la creación de Windows. Nos permite trabajar con unidades físicas, como pulgadas o milímetros, y usar la orientación del espacio de la pantalla o el papel, en el que las *y* crecen hacia abajo, o la orientación matemática tradicional, en el que las *y* crecen hacia arriba.



Transformaciones

Ya hemos mencionado que las transformaciones y el espacio de coordenadas del mundo son elementos nuevos dentro del API de Win32. De modo que estas características no funcionan con versiones previas a Windosw NT, ni siquiera con Windows 95.

Las transformaciones usan un par de fórmulas sencillas para realizar el cambio de coordenadas del espacio del mundo al espacio de página. Si (x,y) son las coordenadas en el espacio del mundo, y (x',y') son las coordenadas en el espacio de página, las fórmulas para obtener estas coordenadas son:

$$x' = x * eM_{11} + y * eM_{21} + eD_x$$
$$y' = x * eM_{12} + y * eM_{22} + eD_y$$

Estas fórmulas se pueden expresar mediante cálculo matricial:

$$|x'\ y'\ 1| = |x\ y\ 1| \cdot \begin{vmatrix} eM_{11} & eM_{12} & 0 \\ eM_{21} & eM_{22} & 0 \\ eD_x & eD_y & 1 \end{vmatrix}$$

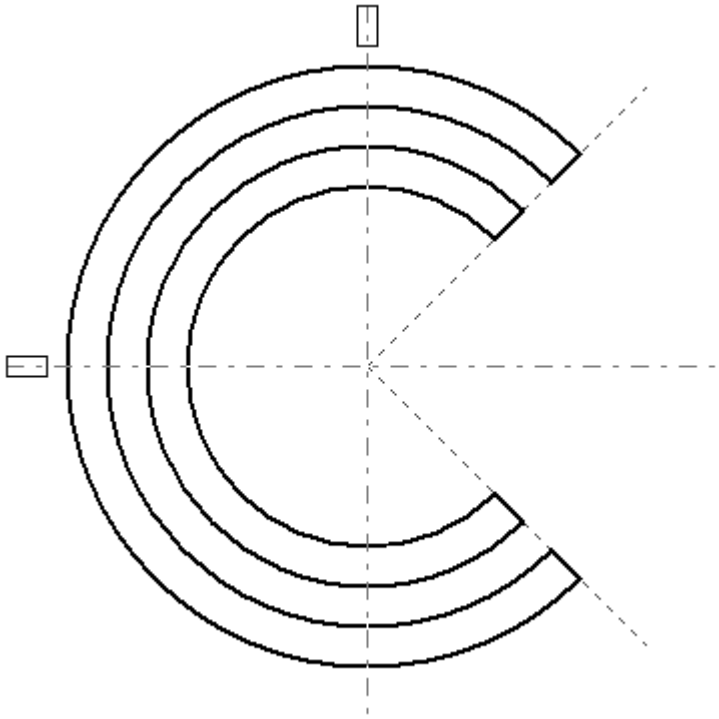
La tercera matriz es la **matriz de transformación**. Esta matriz se maneja en el API mediante una estructura XFORM, y como los valores de la tercera columna son conocidos, no se guardan.

Si no modificamos la matriz de transformación, el sistema usa una **matriz identidad**, en la que todos los elementos son nulos, excepto la diagonal principal:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Si aplicamos esta matriz de transformación, no se produce ninguna transformación:

$$x' = x * eM_{11} + y * eM_{21} + eD_x = x*1 + y*0 + 0 = x$$
$$y' = x * eM_{12} + y * eM_{22} + eD_y = x*0 + y*0 + 0 = y$$



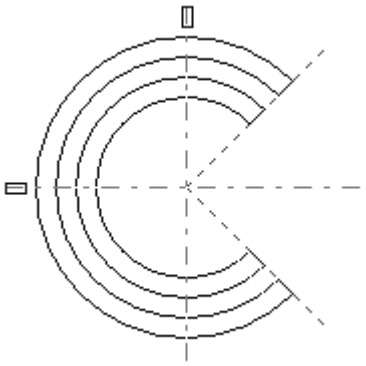
Traslaciones

Si analizamos cada término, veremos que eD_x y eD_y nos permiten hacer traslaciones, es decir, mover puntos en cualquier dirección. (eD_x,eD_y) es, sencillamente, un desplazamiento.

Cambio de escala

Si los términos eM_{21} y eM_{12} son nulos, podemos ver que eM_{11} y eM_{22} realizan un cambio de escala.

```
| 0.5 0   0 |
| 0   0.5 0 |
| 0   0   1 |
```



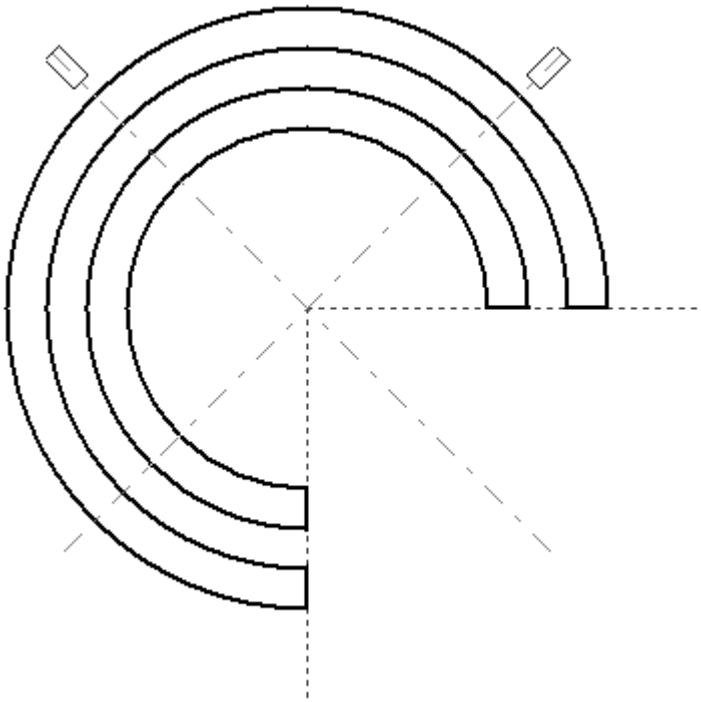
Rotaciones

Las rotaciones implican algunos cálculos trigonométricos sencillos, para rotar la figura un ángulo α, se aplica la siguiente fórmula:

```
x' = x * cos(α) + y * sen(&alpha)
y' = x * (-sen(α)) + y * cos(α)
```

De donde se deduce que:

- eM_{11} es el coseno de α
- eM_{12} es el seno de α
- eM_{21} es menos el seno de α
- eM_{22} es el coseno de α

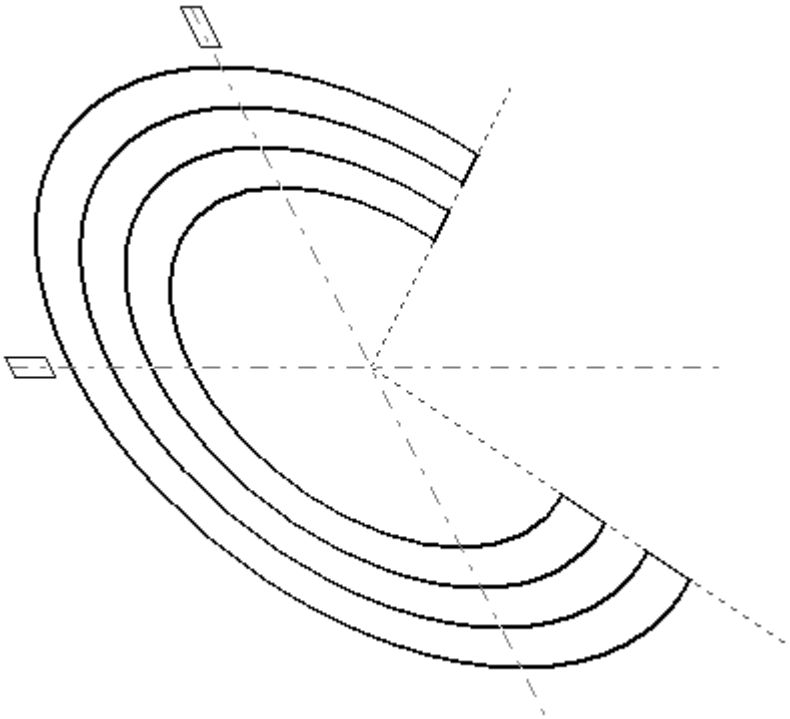


Nota: cuidado con los cálculos, hay que tener en cuenta que en C y C++ se usan ángulos expresados en radianes, 360º son 2π radianes (90º son π/2 radianes , y 45º son π/4 radianes). Además, las funciones trigonométricas están declaradas en el fichero de cabecera *math.h*, y se llaman *sin* y *cos*.

Cambio de ejes

Ponemos inclinar los ejes, de forma que la transformación no sea ortogonal (es decir, que los ejes no sean perpendiculares). Bastará considerar los factores eM_{12} y eM_{21} como constantes de proporcionalidad, horizontal y vertical, respectivamente.

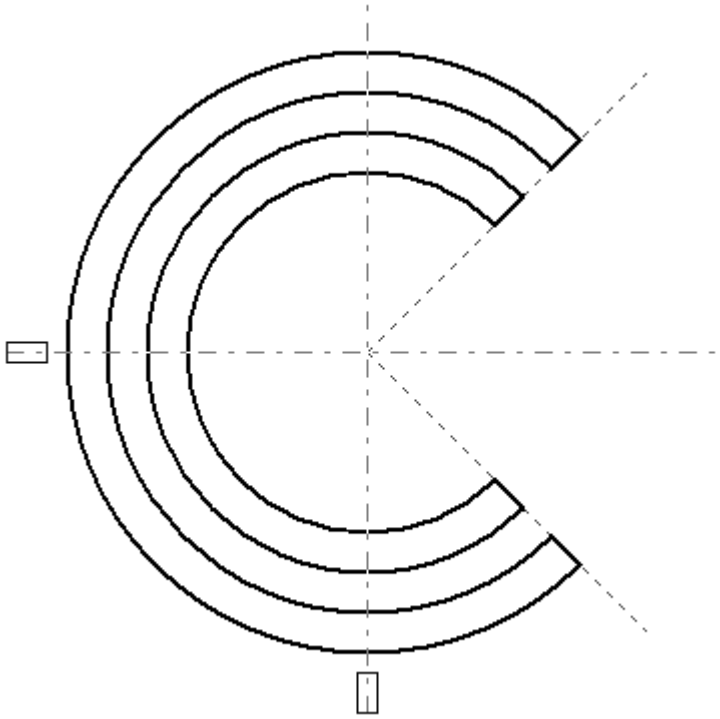
```
x' = x + y * eM21
y' = x * eM12 + y
```



Reflexiones

Podemos cosiderar una reflexión como un caso particular de cambio de escala. Si cambiamos el signo de eM_{11} obtendremos una reflexión en el eje *y*, si cambiamos el signo de eM_{22} obtendremos una reflexión en el eje *x*.

```
x' = x * eM11
y' = y * eM22
```



Aplicar transformaciones

En el API32 hay dos modos gráficos diferentes. El compatible y el avanzado.
El compatible es el modo original de Windows 3.1 y Windows 95, y el único que existía en esos sistemas.
El modo avanzado sólo existe en el API32, para Windows NT y sistemas posteriores, como ME, 2000 y XP. Sólo en este modo es posible usar transformaciones, de modo que antes de aplicar cualquier transformación, será necesario activar el modo avanzado.
Para cambiar el modo gráfico se usa la función SetGraphicsMode, el modo gráfico se aplica a un DC, así que necesitamos un manipulador de DC. Disponemos de dos constantes para activar cada uno de los modos: GM_COMPATIBLE y GM_ADVANCED.
Una vez activado el modo avanzado, ya podemos usar transformaciones. Si no aplicamos ninguna, por defecto se usa la transformación identidad. Y si queremos activar el modo compatible, es imprescindible que la transformación actual sea la de identidad, de otro modo la función SetGraphicsMode fallará.
Para aplicar una transformación usaremos la función SetWorldTransform, que requiere un manipulador de DC y un puntero a una estructura XFORM con la transformación a aplicar.

```
    XFORM xform =   ; // Sin transformar (matriz
identidad)
...
        case WM_PAINT:
           hdc = BeginPaint(hwnd, &ps);
```

```
SetGraphicsMode(hdc, GM_ADVANCED);
SetWorldTransform(hdc, &xform);
// Funciones de trazado gráfico
EndPaint(hwnd, &ps);
break;
```

Combinar transformaciones

No tenemos que limitarnos a hacer transformaciones simples, podemos combinarlas para crear transformaciones complejas, de modo que podemos rotar, trasladar, cambiar ejes, escalar y reflejar mediante una única transformación.
Para ello disponemos de dos opciones diferentes:
Combinar dos transformaciones mediante la función CombineTransform. Esta función obtiene una transformación a partir de otras dos, el resultado de aplicar la transformación obtenida equivale a aplicar las dos transformaciones, una a continuación de la otra.
Modificar la transformación del mundo actual mediante la función ModifyWorldTransform. Mediante esta función será posible combinar la transformación actual con otra, o bien, asignar la matriz de transformación identidad (si se usa el valor MWT_IDENTITY como tercer parámetro. Este tercer parámetro admite otros dos valores: MWT_LEFTMULTIPLY y MWT_RIGHTMULTIPLY, que permite multiplicar la transformación indicada por la izquierda o por la derecha. El resultado puede ser diferente, ya que la multiplicación de matrices no posee la propiedad conmutativa.
Si necesitamos obtener la matriz de transformación actual, podemos hacer uso de la función GetWorldTransform.

Cambios de escala y plumas

Cuando se usan plumas cosméticas para trazar figuras, las únicas que hemos usado hasta ahora, el grosor del trazo se expresa en unidades lógicas, es decir, si duplicamos la escala, el grosor de las líneas se duplica. Esto es así salvo que indiquemos un grosor 0. En ese caso, las líneas siempre son de un pixel de ancho, y por lo tanto, el resultado es que parece que las líneas son más finas cuanto más grande sea la escala. Este efecto puede ser útil cuando trazamos ejes o líneas de ayuda.

Ventanas y viewports

La ventana define en el espacio de coordenadas de la página, mediante dos parámetros: la extensión y el origen.
El *viewport* define el espacio de coordenadas del dispositivo, mediante los mismos parámetros que la ventana: la extensión, y el origen.
En el caso del viewport, al definir el espacio del dispositivo, los valores se expresan en coordendas de dispositivo, es decir, en pixels.

Los puntos en el espacio de página se expresan en coordenadas lógicas, y por lo tanto, también se usan valores lógicos en la ventana. Tanto la extensión como el origen de la ventana se expresa en valores lógicos.
No debemos confundir el espacio de página con la ventana física de la aplicación, aunque existe cierta analogía, la ventana de la que hablamos ahora es un concepto de espacio gráfico, no siempre ligado a la ventana que se muestra en el monitor, es más bien, una ventana que nos permite ver parte del espacio gráfico total.

Extensiones

En el caso de la ventana la extensión se ajusta por la función SetWindowExtEx. También existe una función para obtener ese parámetro: GetWindowExtEx.
Para le *viewport* también existe una pareja de funciones para asignar y leer la extensión: SetViewportExtEx y GetViewportExtEx.

Orígenes

Tanto en el caso de la ventana como en el del viewport, podemos definir un origen de coordenadas, en el caso de la ventana se usa la función SetWindowOrgEx y en el caso del viewport, la función SetViewportOrgEx. Por supuesto, existen las funciones simétricas, para leer esas coordenadas: GetWindowOrgEx y GetViewportOrgEx.

Mapeos

Podríamos titular este apartado como transformaciones del espacio de página al de dispositivo.
Para realizar estas transformaciones se usan los valores de extensión y origen de la ventana y del viewport. Los puntos situados en la ventana se proyectan (o se mapean) al espacio del viewport. Las fórmulas para hacer esas proyecciones son simples:

$$D_x = ((L_x - WO_x) * VE_x / WE_x) + VO_x$$
$$D_y = ((L_y - WO_y) * VE_y / WE_y) + VO_y$$

Donde:

- (D_x,D_y) son las coordenadas del punto en unidades de dispositivo.
- (L_x,L_y) las coordenadas del puntir en unidades lógicas (unidades del espacio de página).
- (WO_x,WO_y) las coordenadas del origen de la ventana.
- (VO_x,VO_y) las coordenadas del origen del viewport.
- (WE_x,WE_y) es la extensión de la ventana.
- (VE_x,VE_y) la extensión del viewport.

Básicamente, estas fórmulas definen un cambio de escala, y una traslación.
Existen dos funciones para realizar estos cálculos, así como sus inversos. Podemos, de este modo, pasar las coordenadas de un punto de un espacio al otro. Para pasar un punto en coordendadas lógicas (de página) a coordenadas de dispositivo, se usa la función LPtoDP. Para pasar de coordenadas de dispositivo a coordenadas lógicas, se usar la función DPtoLP.

Modos de mapeo predefinidos

En Windows existen ocho posibles modos de mapeo, cada uno con sus características propias:

- Isotrópico: el mapeo entre el espacio de página y el del dispositivo se define por la aplicación, cambiando las extensiones y orígenes de la ventana y del *viewport*. Las medidas en ambos ejes son iguales, pero la orientación se puede definir por la aplicación.
- No isotrópico: igual que el anterior, pero las medidas en el eje x no tienen por qué ser iguales que en el eje y.
- Inglés alto: cada unidad del espacio de página se mapea a 0.001 pulgadas en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Inglés bajo: cada unidad del espacio de página se mapea a 0.01 pulgadas en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Métrico alto: cada unidad del espacio de página se mapea a 0.01 milímetros en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Métrico bajo: cada unidad del espacio de página se mapea a 0.1 milímetros en el espacio de dispositivo. Las x crecen hacia la derecha, las y hacia arriba.
- Texto: cada unidad del espacio de página se mapea a un pixel, es decir, no se hace ningún cambio de escala.
- Twips: cada unidad del espacio de página se mapea a 1/20 de punto de impresora (un twip), que equivale a 1/1440 de pulgada. Las x crecen hacia la derecha, las y hacia arriba.

Para activar un modo de mapeo se usa la función SetMapMode. Esta función precisa dos parámetros: un manipulador de DC, y el identificador del modo de mapeo (MM_ISOTROPIC, MM_ANISOTROPIC, MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, MM_TEXT y MM_TWIPS, respectivamente). Para averiguar el modo actual GetMapMode.

En el caso de los seis últimos modos (todos menos el isotrópico y el no isotrópico), la extensión del viewport no puede alterarse, ya que se ajusta automáticamente, en función de la extensión de la ventana.

Cuando se usan los modos isotrópico o no isotrópico, es necesario ajustar la extensión y origen del *viewport*, ya que en estos modos, el cambio de escala se define por la aplicación. En el caso del modo isotrópico es importante ajustar la extensión de la ventana antes de hacerlo con el *viewport*.

Modo por defecto

El modo por defecto es el de texto, que además es el único que es dependiente del dispositivo.

Este modo es útil si la salida sólo se va a visualizar en pantalla, pero no resultará muy conveniente si tenemos que asegurar que el tamaño de nuestros gráficos es real (por ejemplo si dibujamos un cuadrado de 10 centímetros de lado), y mucho menos si tenemos que realizar salidas a impresora, si trazamos un cuadrado de 100 pixels en pantalla, el tamaño en impresora dependerá mucho de los puntos por pulgada de resolución que tenga la impresora.

En cada caso será interesante escoger un modo independiente de dispositivo, ya sea en pulgadas, milímetros, o puntos de impresora.

Transformaciones definidas por el usuario

Hay dos modos de mapeo que implican que la transformación se difine por el usuario: el isotrópico y el no isotrópico. Son modos muy similares, pero el isotrópico asegura que las unidades lógicas son del mismo tamaño en el eje x y el en eje y. El modo no isotrópico permite usar unidades diferentes en cada eje.
Por ejemplo, para obtener una resolución de 1/3 milímetro, podemos usar el modo isotrópico de esta manera:

```
SetMapMode(hDC, MM_ISOTROPIC);
SetWindowExtEx(hDC, 3*GetDeviceCaps(hDC, HORZSIZE),
                    3*GetDeviceCaps(hDC,   VERTSIZE),
NULL);
SetViewportExtEx(hDC, GetDeviceCaps(hDC, HORZRES),
                      GetDeviceCaps(hDC,    VERTRES),
NULL);
```

Invocamos la función GetDeviceCaps para obtener las dimensiones del dispositivo en milímetros (HORZSIZE,VERTSIZE), y el pixels (HORZRES,VERTRES). Hacemos que la extensión x e y de la ventana sea el triple del tamaño del dispositivo en milímetros, y que la extensión del *viewport* sea el tamaño del dispositivo en pixels. Esto hace que cada unidad de página (unidad lógica) se convierta en 1/3 milímetro.

Modos gráficos y sentido de los arcos

Hay que tener en cuenta que el sentido en que se trazan los arcos es importante, y que el resultado de la salida gráfica dependerá del modo gráfico y del modo de mapeo seleccionados, y en el caso de los modos isotrópico y no isotrópico, de los signos de los valores de las extensiones de ventana y *viewport*.
En el modo gráfico avanzado, los arcos siempre se trazan en el sentido de las agujas del reloj en el espacio lógico. Esto independiza la salida del modo de mapeo elegido. En el caso del modo gráfico compatible se usa el sentido de trazado actual de los arcos para trazarlos en el espacio del dispositivo, es decir, una vez aplicado el mapeo. De modo que si cambiamos el modo de mapeo, el aspecto de los arcos puede cambiar.
Recordemos que podemos cambiar el sentido de trazado de los arcos en el espacio de dispositivo, usando la función SetArcDirection.

Nota: siempre que podamos, usaremos el modo gráfico avanzado, ya que no sólo nos permite usar transformaciones, sino que además simplifica el trazado de arcos, al no tener que preocuparse del modo de mapeo que se use en cada caso.

Otras funciones

Hay algunas funciones más relacionadas con espacios de coordenadas:

ClientToScreen	Convierte coordenadas de cliente a coordenadas de pantalla.
ScreenToClient	Convierte coordenadas de pantalla a coordenadas de cliente.
GetCurrentPositionEx	Recupera la posición actual del cursor gráfico en coordenadas lógicas.
OffsetWindowOrgEx	Desplaza el origen de la ventana en las cantidades especificadas.
OffsetViewportOrgEx	Desplaza el origen del *viewport* en las cantidades especificadas.
ScaleWindowExtEx	Modifica la extensión de la ventana en el factor especificado.
ScaleViewportExtEx	Modifica la extensión del *viewport* en el factor especificado.

Capítulo 29 Objetos básicos del GDI: Plumas geométricas

En el capítulo 18 vimos cómo crear y usar plumas cosméticas, y hablamos un poco de plumas geométricas, aunque sin entrar en detalles. En este capítulo veremos cómo crear, usar y destruir plumas geométricas, así como sus características y propiedades.

Atributos de las plumas geométricas

Las plumas geométricas tienen siete atributos: anchura, estilo, color, patrón, rayado, estilo de extremos y estilo de uniones. Recordemos que las plumas cosméticas sólo tenían los tres primeros.

Los patrones y el rayado son atributos que hasta ahora sólo hemos asociado a pinceles, pero también los poseen las plumas geométricas.

Esto hace que las plumas geométricas sean más difíciles de crear, y más lentas a la hora de trazar líneas, pero son mucho más potentes y versátiles que las cosméticas.

Para crear plumas geométricas usaremos la función ExtCreatePen.

Veamos ahora algunos de esos atributos en detalle, y qué opciones existen.

Anchura

Vimos que en el caso de las plumas cosméticas, el grosor se expresaba en unidades de dispositivo, es decir, en pixels. En las geométricas se expresa en unidades lógicas, es decir, su anchura se ve afectada por las transformaciones del sistema de coordenadas.

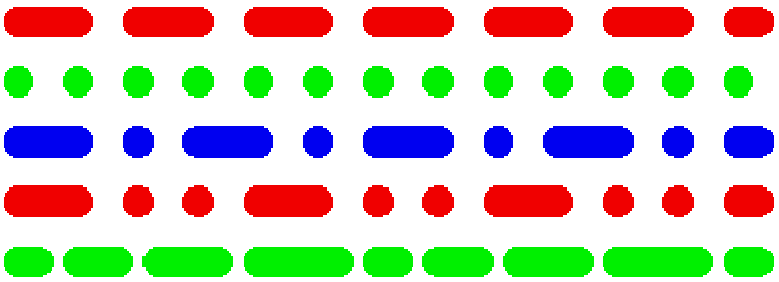
Aunque solemos decir que las funciones CreatePen y CreatePenIndirect crean plumas cosméticas, esto es falso. Windows sólo crea plumas cosméticas de un pixel (cuando indicamos un grosor de cero en estas funciones), de modo que si creamos plumas más anchas usando estas funciones, en realidad estaremos creando plumas geométricas. Esta limitación de Windows no tiene por qué respetarse en futuras versiones, de modo que estas funciones, podrían crear plumas cosméticas de más de un pixel en el futuro, así que seguiremos diciendo que las plumas creadas usando estas dos funciones son cosméticas, aunque el grosor sea distinto de cero. Pero debemos tener en cuenta que estas plumas se verán afectadas por las transformaciones del espacio del mundo, o por el mapeo.

Estilo de línea

El estilo de línea define el tipo de trazos de las líneas. Recordemos los posibles estilos de línea:

Estilo	Descripción
Sólido	Las líneas serán continuas y sólidas.
Trazos	Líneas de trazos.
Puntos	Líneas de puntos.
Trazo y punto	Líneas alternan puntos y trazos.
Trazo, punto, punto	Líneas alternan líneas y dobles puntos.
Nulo	Las líneas son invisibles.
Dentro de marco	Las líneas serán sólidas. Sólo en el caso de plumas geométricas, y cuando se usan con funciones que requieran un rectángulo que sirva como límite, las dimensiones de la figura se reducirán para que se ajusten por completo al interior del rectángulo, teniendo en cuenta el grosor de la pluma.

En el caso de las plumas geométricas, los estilos no sólidos no están limitados a un pixel, como en el caso de las plumas cosméticas. Además, existe otro estilo posible, en el que el usuario puede definir las longitudes de los trazos y las separaciones.



Color

Especifica el color de la pluma. Se puede usar una estructura COLORREF para indicar el color.

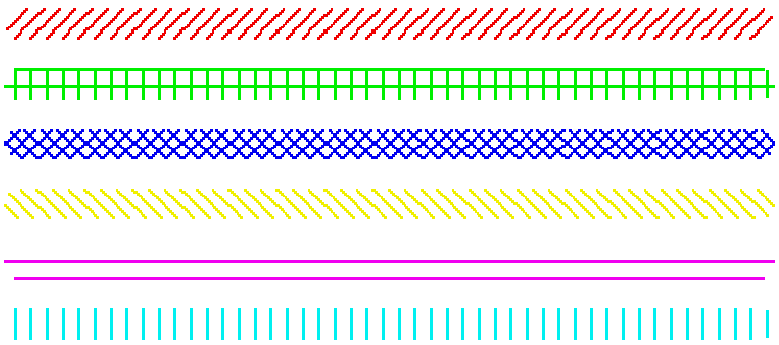
Patrón

Especifica un patrón de mapa de bits que se repite en el trazo. Uno de los patrones es el de rayado, que puede ser cualquiera de los seis predefinidos. Pero también se pueden usar patrones creados a partir de cualquier mapa de bits de 8x8 pixels, patrones vacíos o sólidos.

Rayado

El rayado es un tipo particular de patrón. Normalmente nos referiremos a los rayados predefinidos, que son seis, como vimos en el capítulo de plumas:

Valor	Significado
Diagonal descendente	Trama de líneas diagonales a 45º descendentes de izquierda a derecha.
Cruz	Trama de líneas horizontales y verticales.
Cruz diagonal	Trama de líneas diagonales a 45º cruzadas.
Diagonal ascendente	Trama de líneas diagonales a 45º ascendentes de izquierda a derecha.
Horizontal	Trama de líneas horizontales.
Vertical	Trama de líneas verticales.



Estilo de final (tapón)

Los extremos de las líneas pueden ser de diferentes tipos. Para las plumas geométricas tenemos tres posibles valores:

Valor	Significado
Redondeado	Las líneas terminan con un semicírculo.
Cuadrado	Las líneas terminan con medio cuadrado.
Plano	Las líneas terminan de forma abrupta.

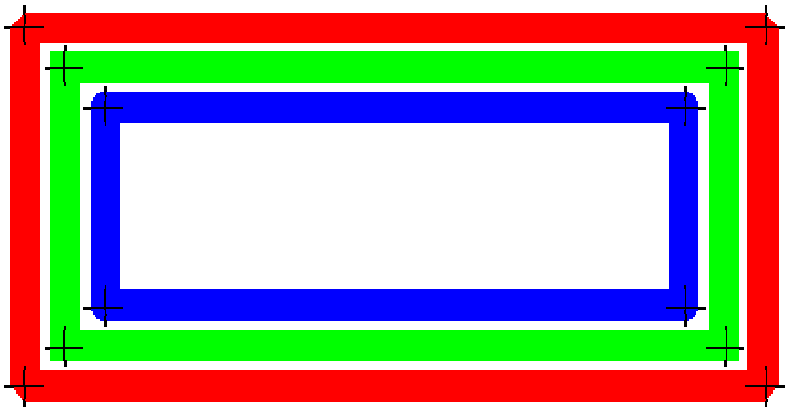
Por ejemplo, estas líneas fueron trazadas con estos tres estilos de final.

Estilo de unión

Para las uniones de distintas líneas de una figura cerrada también disponemos de tres estilos. Estos estilos no se aplican a líneas unidas que no formen parte de una línea poligonal o de una figura cerrada:

Valor	Significado
Redondeado	Las esquinas se redondean.
Picudas (miter)	Las esquinas se dejan sin recortar ni redondear.
Biseladas (Bevel)	Las puntas de las esquinas se cortan a bisel.

Por ejemplo, estas figuras fueron trazadas con estos tres estilos de final. La azul con el estilo redondeado, la verde picuda y la roja viselada:



Crear una pluma geométrica

Para crear plumas geométricas se usa la función ExtCreatePen:

```
   HPEN pluma;
   LOGBRUSH lb = {BS_SOLID, RGB(240, 0, 0), 0};

   pluma = ExtCreatePen(
      PS_GEOMETRIC  |  PS_DASH  |  PS_ENDCAP_ROUND  |
PS_JOIN_ROUND,
      15, &lb, 0, NULL);
```

Como estas plumas pueden usar atributos de pincel para el trazado, necesitamos suministrar una estructura LOGBRUSH para definir el estilo y color de la pluma.

Seleccionar una pluma geométrica

No hay diferencia, a la hora de usarlas, entre las plumas geométricas y las cosméticas. En ambos casos, y como sucede con el resto de los objetos del GDI, se usa la función SelectObject:

```
   SelectObject(hdc, pluma);
   MoveToEx(hdc, 30, 30, NULL);
```

```
LineTo(hdc, 400, 30);
```

Destruir una pluma geométrica

Y, por supuesto, cuando ya no necesitemos una pluma geométrica, deberemos destruirla y liberar los recursos que consume, usando la función DestroyObject:

```
DeleteObject(pluma);
```

Capítulo 30 Objetos básicos de usuario: El Caret

Los carets son las marcas intermitentes que nos indican dónde se insertará el texto o los graficos cuando un usuario los introduzca. Normalmente son pequeñas líneas verticales u horizontales, aunque pueden ser rectángulos de distintos tonos o incluso mapas de bits.
Ya sabemos que sólo una ventana puede tener el foco en un determinado momento, de modo que sólo puede mostrarse un caret en un momento determinado, precisamente en la ventana que tiene el foco.
Veremos en este capítulo como crear, modificar, ocultar o mostrar carets.

Recibir y perder el foco

El caret es un recurso compartido, es decir, sólo existe un caret en todo el sistema. Esto implica que debemos tener en cuenta algunos aspectos importantes. Por ejemplo, si nuestra aplicación usa un caret, debe crearlo cada vez que recibe el foco y destruirlo cuando lo pierde, de modo que la nueva aplicación que recibe el foco pueda crear el suyo.
Es importante por lo tanto, saber cuándo nuestra aplicación recibe o pierde el foco, para ello disponemos de dos mensajes muy útiles: WM_SETFOCUS se recibe justo cuando la aplicación recibe el foco, y WM_KILLFOCUS se recibe justo antes de perderlo.
Las aplicaciones que trabajen con carets deben procesar estos dos mensajes. Cuando se recibe WM_SETFOCUS se crea y muestra el caret y cuando se recibe WM_KILLFOCUS se destruye.

Crear y destruir carets

Para crear un caret usaremos la función CreateCaret, que puede crear cualquiera de los carets posibles: verticales, horizontales, sólidos, grises y de mapas de bits.
Esta función precisa de cuatro parámetros. El primero es el manipulador de ventana para la que creamos el caret. El segundo es un manipulador de mapa de

bits, si es NULL se creará un caret sólido, si es 1 el caret será gris (trama de puntos), o puede ser un mapa de bits que se usará como figura del caret. El tercer parámetro es la anchura, en unidades lógicas, y el cuarto la altura.
Hay que tener en cuenta que las transformaciones y mapeos afectan al tamaño del caret.
Por ejemplo, para crear un caret vertical:

```
        case WM_SETFOCUS:
           CreateCaret(hwnd, (HBITMAP)NULL, 0, 20);
```

Por ejemplo, para crear un caret a partir de un mapa de bits:

```
        case WM_SETFOCUS:
           CreateCaret(hwnd, hBitmap, 0, 0);
```

Para destruirlo, cuando perdemos el foco usaremos la función DestroyCaret:

```
        case WM_KILLFOCUS:
           DestroyCaret();
           break;
```

Mostrar y ocultar carets

Crear un caret no lo muestra automáticamente, es necesario mostrarlo explícitamente mediante la función ShowCaret. Esto completa la respuesta al mensaje de activación del foco:

```
        case WM_SETFOCUS:
           CreateCaret(hwnd, (HBITMAP)NULL, 0, 20);
           ShowCaret(hwnd);
           break;
```

Por otra parte, a veces es necesario ocultar el caret, en particular, cuando se actualiza la pantalla, de modo que no interfiera con el contenido de la ventana. Para ocultar el caret se usa la función HideCaret.

Procesar mensajes WM_PAINT

Si no tenemos la precaución de ocultar el caret durante la actualización de la ventana, éste puede interferir con el nuevo contenido, de modo que se corrompa parte del contenido. Cuando nuestra aplicación trabaje con carets y procesemos el mensaje WM_PAINT, debemos ocultar el caret mientras duran la operaciones de actualización:

```
        case WM_PAINT:
```

```
            HideCaret(hwnd);
            hdc = BeginPaint(hwnd, &ps);
            ActualizarPantalla(hdc);
            EndPaint(hwnd, &ps);
            ShowCaret(hwnd);
            break;
```

Cambiar posición de un caret

Otro parámetro importante con respecto a los carets es su posición en pantalla, ya que se usa para indicar el punto de inserción de texto o de gráficos, normalmente la posición se modificará con mucha frecuencia.
Para cambiar la posición del caret se usa la función SetCaretPos, si necesitamos obtener la posición actual, podemos usar GetCaretPos. La posición del caret se puede modificar aunque el caret esté oculto.

```
            SetCaretPos(120,120);
```

Cambiar velocidad de parpadeo de un caret

El caret tiene otra propiedad importante: el parpadeo. La velocidad del parpadeo se puede modificar usando el Panel de Control, pero el API también proporciona una función para modificar esa velocidad, aunque es nuestra responsabilidad que ese cambio sea a petición del usuario, ya que al tratarse de un recurso compartido, todas las aplicaciones pueden verse afectadas por este cambio.
Para asignar un nuevo valor al parpadeo se usa la función SetCaretBlinkTime y para obtener la velocidad actual, GetCaretBlinkTime

```
            ActualBlink = GetCaretBlinkTime();
            SetCaretBlinkTime(50);
```

Capítulo 31 Objetos básicos del usuario: El icono

Le toca el turno a otro recurso muy familiar para el usuario de Windows: el icono.
Un icono es un pequeño mapa de bits, combinado con una máscara que permite que parte de él sea transparente. El resultado es que las figuras representadas por iconos pueden tener diferentes formas, y no tienen por qué ser rectangulares.
Los usamos como ayuda para representar objetos: directorios, ficheros, aplicaciones, etc.

Punto activo

Los iconos tienen un punto activo (hot spot), como veremos que sucede también con los cursores. En el caso de los iconos, ese punto suele ser el centro, y se usa para tareas de alineación y espaciado de iconos.

Tamaños

En Windows se usan iconos en dos entornos, el sistema y el *shell*, y para cada uno de ellos usamos dos tamaños de icono, el grande y el pequeño.
El icono de sistema pequeño se usa en las barras de título de las ventanas. Para averiguar el tamaño de este icono hay que llamar a GetSystemMetrics con SM_CXSMICON y SM_CYSMICON.
El icono de sistema grande es que se usa normalmente en las aplicaciones, las funciones DrawIcon y LoadIcon usan por defecto, estos iconos. Para averiguar el tamaño de este icono se puede llamar a GetSystemMetrics con SM_CXICON y SM_CYICON.
El icono pequeño del shell se usa en el explorador de windows y en los diálogos comunes.
El icono grande del shell se usa en el escritorio.
Cuando creemos iconos a medida en nuestras aplicaciones, deberemos suministrar recursos de icono de los siguientes tamaños:

- 48x48, 256 colores
- 32x32, 16 colores
- 16x16 pixels, 16 colores

Asociar iconos a una aplicación

Podemos decidir qué iconos usará nuestra aplicación para la barra de título, y qué icono se asocia a la aplicación al registrar la clase de ventana.
Cuando se rellena la estructura WNDCLASSEX que se usa para registrar nuestra clase de ventana, el miembro *hIcon* debe ser un icono de 32x32 y el miembro *hIconSm* uno de 16x16. Aunque ya hemos visto en algunos ejemplos que esto es opcional. Si usamos un icono de 32x32 para el *hIconSm* el sistema lo escalará automáticamente a 16x16.

```
    wincl.hIcon    =    LoadIcon    (hThisInstance,
"tajmahal");
    wincl.hIconSm    =    LoadIcon    (hThisInstance,
"lapiz");
```

Tipos

Existen iconos estándar disponibles para cualquier aplicación, y también es posible modificar las imágenes asociadas a esos iconos estándar, personalizando el escritorio de Windows.

Los iconos estándar son:

Icono	Identificador	Descripción
Aplicación	IDI_APPLICATION	Icono de aplicación por defecto.
Información	IDI_ASTERISK	Asterisco (usado en mensajes de información).
Exclamación	IDI_EXCLAMATION	Signo de exclamación (usado en mensajes de aviso).
Aviso importante	IDI_HAND	Icono de mano extentida (usado en mensajes de aviso importantes).
Interrogación	IDI_QUESTION	Signo de interrogación (usado en mensajes de petición de datos).
Logo	IDI_WINLOGO	Icono de logo de Windows.

Cargar uno de estos iconos es sencillo, basta usar la función LoadIcon, indicando como manipulador de instancia el valor NULL, y como identificador de icono, el que queramos cargar:

```
HICON icono = LoadIcon(NULL, IDI_EXCLAMATION);
```

Por otra parte, en nuestras aplicaciones podremos crear nuestros propios iconos, bien a partir de ficheros de recursos, o directamente a partir de datos binarios.
Generalmente usaremos ficheros de recursos, ya que crear iconos durante la ejecución hace que estos sean dependientes del dispositivo, y su aspecto puede ser impredecible.
En el caso de iconos de medidas normales, de 32x32 o de 16x16, se usa la función LoadIcon, en el caso de iconos de 48x48 se debe usar LoadImage:

```
   HICON icono1, icono2, icono3;

   icono1    =    LoadImage(hInstance,    "tajmahal",
IMAGE_ICON, 0, 0, LR_LOADREALSIZE);
   icono2 = LoadIcon(hInstance, "antena");
   icono3 = LoadIcon(hInstance, "lapiz");
```

Iconos en ficheros de recursos

Existen programas de edición de iconos en Internet, como por ejemplo IconEdit2, que es ShareWare. (Hay que registrarse para poder guardar los ficheros de iconos). También podemos optar por usar uno de los muchos iconos de galería que se pueden encontrar en Internet.
Se pueden incluir iconos diseñados por nosotros en el fichero de recursos mediante la sentencia ICON, y obtener un manipulador para ellos usando las funciones LoadIcon o LoadImage.

```
antena ICON "station.ico"
```

```
lapiz ICON "fayDrafts.ico"
tajmahal ICON "tajmahal.ico"
```

Iconos en controles estáticos

Ya hemos usado iconos anteriormente como parte de los controles estáticos en el capítulo 10.

Mostrar iconos

En futuros capítulos veremos cómo incluirlos en menús o botones, de momento nos conformaremos con mostrarlos en pantalla.
Se puede obtener información mediante GetIconInfo, y mostrar el icono mediante DrawIconEx o DrawIcon. Si se usa el parámetro DI_COMPAT se mostrará la imagen por defecto, si no, se mostrará la imagen que indiquemos. La función DrawIconEx es más potente en el sentido de que nos permite mostrar iconos de tamaños diferentes de 32x32, y además nos permite escalarlos.

```
            DrawIconEx(hdc, 10, 10,
               LoadImage(hInstance,       "tajmahal",
IMAGE_ICON, 0, 0, LR_LOADREALSIZE),
               0, 0, 0, NULL, DI_NORMAL);
            DrawIcon(hdc,   60,10,LoadIcon(hInstance,
"antena"));
            DrawIconEx(hdc,          100,          10,
LoadIcon(hInstance, "lapiz"),
               16, 16, 0, NULL, DI_NORMAL);
            DrawIcon(hdc,  140,  10,  LoadIcon(NULL,
IDI_APPLICATION));
```

Destrucción de iconos

DestroyIcon sólo se puede aplicar a iconos creados mediante CreateIconIndirect. No recomiendo usar este tipo de iconos, ya que son dependientes del dispositivo, y pueden producir resultados no deseados en algunos equipos.

Capítulo 32 Objetos básicos del usuario: El cursor

El cursor o puntero, indica la posición del ratón en pantalla, y nos permite acceder a los elementos de las ventanas, al tiempo que su aspecto nos da pistas sobre la acción asociada al cursor, o sobre el estado del sistema.
Es un recurso importante y único en el sistema, por lo que hay que compartirlo entre las diferentes aplicaciones. Algunos de los cambios que hagamos en el cursor se deben deshacer cuando el control pase a otras aplicaciones.

Cursor de clase

Uno de los parámetros que asignamos al registrar una clase de ventana, usando la estructura WNDCLASS o WNDCLASSEX y las funciones RegisterClass o RegisterClassEx es, precisamente, el cursor de la clase. Windows siempre muestra ese cursor mientras esté dentro del área de cliente de la ventana.

```
    WNDCLASSEX wincl;         /* Estructura de datos
para la clase de ventana */

    /* Estructura de la ventana */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = "NUESTRA_CLASE";
    wincl.lpfnWndProc  =  WindowProcedure;         /*
Esta función es invocada por Windows */
    wincl.style = CS_DBLCLKS;                      /*
Captura los doble-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Usar icono y puntero por defector */
    wincl.hIcon = LoadIcon (hThisInstance, "Icono");
    wincl.hIconSm    =    LoadIcon    (hThisInstance,
"Icono");
    wincl.hCursor = LoadCursor (NULL, IDC_ARROW);
    wincl.lpszMenuName = "Menu";
    wincl.cbClsExtra = 0;                          /*
Sin información adicional para la */
    wincl.cbWndExtra = 0;                          /*
clase o la ventana */
    /* Usar el color de fondo por defecto para es
escritorio */
    wincl.hbrBackground
=GetSysColorBrush(COLOR_BACKGROUND);

    /*  Registrar  la  clase  de  ventana,  si  falla,
salir del programa */
```

```
    if(!RegisterClassEx(&wincl)) return 0;
```

Hasta ahora, en todos nuestros ejemplos cargábamos el cursor de la flecha, pero en este capítulo veremos que podemos usar nuestro propio cursor de clase para nuestras ventanas, bastará con asignar otro cursor al miembro *hCursor*. Veremos a continuación qué cursores podemos usar.

Cursores de recursos

Se pueden incluir cursores diseñados por nosotros o almacenados en ficheros ".cur" en el fichero de recursos mediante la sentencia CURSOR, y obtener un manipulador para ellos usando las funciones LoadCursor o LoadImage.

```
Flecha3D CURSOR "3dgarro.cur"
```

Para usar estos cursores en nuestra aplicación usaremos la función LoadCursor, indicando la instancia actual y el identificador de recurso:

```
   HCURSOR    flecha3d    =    LoadCursor(hInstance,
"flecha3D");
```

O bien la funcón LoadImage:

```
HCURSOR flecha3d = LoadImage(hInstance, "flecha3D",
IMAGE_CURSOR, 0, 0, LR_LOADREALSIZE);
```

Cursores estándar

Podríamos llamarlos cursores de stock, aunque en realidad no lo son, ya que los cursores estándar se pueden personalizar por el usuario al cambiar las opciones del escritorio.
Existen los siguientes cursores estándar:

Valor	Descripción
IDC_APPSTARTING	Flecha estándar y un pequeño reloj de arena.
IDC_ARROW	Flecha estándar.
IDC_CROSS	Cruz.
IDC_IBEAM	I para texto.
IDC_ICON	Sólo en Windows NT: icono vacío
IDC_NO	Círculo barrado.
IDC_SIZE	Sólo en Windows NT: flecha de cuatro

	puntas.
IDC_SIZEALL	Igual que IDC_SIZE.
IDC_SIZENESW	Flecha de dos puntas noreste y sudoeste.
IDC_SIZENS	Flecha de dos puntas, norte y sur.
IDC_SIZENWSE	Flecha de dos puntas, noroeste y sudeste.
IDC_SIZEWE	Flecha de dos puntas, este y oeste.
IDC_UPARROW	Flecha vertical.
IDC_WAIT	Reloj de arena.

El aspecto gráfico de cada uno depende de la configuración del escritorio de Windows, y se puede modificar usando el Panel de Control. En nuestra aplicación podemos cargar cualquiera de ellos usando la función LoadCursor, indicando NULL como manipulador de instancia, y el identificador que queramos.

```
   HCURSOR cursor = LoadCursor(NULL, IDC_ARROW);
   DrawIconEx(hdc, 10,120, cursor, 0, 0, 0, NULL,
DI_NORMAL|DI_COMPAT);
```

También podemos cambiar esos cursores mediante la función del API SetSystemCursor. El primer parámetro es un manipulador de cursor, y los valores adecuados para el segundo parámetro son los mismos que en la tabla anterior, pero con el prefijo "OCR_", en lugar de "IDC_":

```
   HCURSOR flecha3d = LoadCursor(hInstance,
"flecha3D");
   SetSystemCursor(flecha3d, OCR_NORMAL);
```

Estos cambios son permanentes, en el sentido de que no se restituyen al abandorar la aplicación, y para recuperar los cursores previos hay que usar la misma función o bien el panel de control.

Similitud entre iconos y cursores

Existe cierta similitud entre cursores e iconos, el formato en el que se guardan es similar, y se puede usar la función DrawIconEx para mostrar un icono, tanto como un cursor. Del mismo modo que se puede usar GetIconInfo para obtener información sobre un cursor.

```
   hdc = BeginPaint(hwnd, &ps);
   DrawIconEx(hdc, 10,50, caballo, 0,0, 0, NULL,
DI_NORMAL|DI_COMPAT);
   EndPaint(hwnd, &ps);
```

Además, es posible usar un icono como cursor, aunque con algunas limitaciones, dependiendo del hardware instalado.

```
   HICON tajmahal = LoadImage(hInstance, "tajmahal",
IMAGE_ICON, 0, 0, LR_LOADREALSIZE);
   SetClassLong(hwnd, GCL_HCURSOR, (LONG)tajmahal);
```

Los cursores pueden ser monocromo, en color, o animados. Aunque algunos sistemas sólo admiten determinados tipos de cursores, monocromo y de determinadas dimensiones, sobre todo antes de windows 95. Por ejemplo, no se pueden usar cursores en color con una pantalla VGA.

El punto activo (Hot Spot)

La similitud entre iconos y cursores se da también en la propiedad del punto activo. En el caso del icono sólo se usaba para alinear o situar el icono en pantalla, en el caso de cursor su función es diferente, el punto activo indica el punto exacto de la acción del ratón, es el punto que se considera como la posición del cursor.
Los mensajes del ratón suelen referirse a un punto concreto, y ese punto es el punto activo del cursor.

Crear cursores

Los cursores estándar no es necesario crearlos, ya que existen como parte del sistema. Podemos usar las funciones LoadCursor o LoadImage para obtener manipuladores de esos cursores. Las mismas funciones se usan para obtener manipuladores de cursores de recursos.
En el caso de cursores animados no es posible crearlos a partir de recursos, de modo que hay que cargarlos directamente desde un fichero ".ani" durante la ejecución, usando la función LoadCursorFromFile, esta función también puede cargar ficheros de cursores no animados, con extensión ".cur".

```
   HCURSOR                caballo                =
LoadCursorFromFile("horse.ani");
```

También se pueden crear de forma dinámica mediante CreateIconIndirect, y una estructura ICONINFO, GetIconInfo, pero es preferible usar cursores de recursos, ya que se evitan problemas derivados de la dependencia de dispositivo.

Posición del cursor

La posición del cursor cambia como reflejo del movimiento del ratón, pero podemos cambiar su posición en cualquier momento usando la función SetCursorPos. También, podemos recuperar la posición actual del cursor mediante GetCursorPos.

Estas dos funciones trabajan con coordenadas de pantalla. Si queremos obtener o usar coordenadas de ventana para leer o modificar la posición del cursor podemos usar las funciones ScreenToClient o ClientToScreen.

```
punto.x = 60;
punto.y = 60;
ClientToScreen(hwnd, &punto);
SetCursorPos(punto.x, punto.y);
GetCursorPos(&punto);
ScreenToClient(hwnd, &punto);
```

Apariencia

Para obtener un manipulador del cursor actual se usa la función GetCursor.
Para cambiar el cursor actual se usa la función SetCursor, y sólo después de mover el cursor se mostrará la nueva apariencia. Recordemos que el sistema se encarga de mostrar siempre el cursor de acuerdo para la zona sobre la que esté, y cuando se sitúa en el área de cliente, se usa el cursor de la clase.
De modo que para que sea posible cambiar la apariencia del cursor, el cursor de la clase debe ser NULL. Pero esto significa que si movemos el cursor fuera del área de cliente, el cursor cambia automáticamente, y al regresar al área de cliente, mantiene el aspecto que tenía después de la última asignación de cursor. Por ejemplo, si situamos el cursor sobre el borde derecho se mostrará el cursor de doble flecha, este-oeste. Si volvemos al área de cliente, se mantiene ese cursor.

```
SetCursor(LoadCursor(hInstance, "flecha3D"));
```

Hay dos modos de evitar esto, uno es modificar el cursor de la clase, el otro, procesar el mensaje WM_SETCURSOR.

Modificar el cursor de clase

Ya vimos al principio del capítulo que es posible asignar un cursor para la clase de ventana, y que ese cursor se usará en el área de cliente de todas las ventanas de esa clase. Ahora bien, es posible cambiar el cursor asociado a una clase, y en consecuencia, para todas las ventanas de dicha clase, esto se hace mediante la función SetClassLong.
Esta función puede, en principio, cambiar cualquier valor de la estructura WNDCLASS o WNDCLASSEX, pero en este caso nos limitaremos al cursor. Es tan sencillo como llamarla, usando como parámetros el manipulador de ventana, el índice correspondiente al cursor, que es GCL_HCURSOR, y el manipulador del nuevo cursor de clase, convertido a un valor LONG:

```
SetClassLong(hwnd, GCL_HCURSOR,
   (LONG)LoadCursorFromFile("horse.ani"));
```

El mensaje WM_SETCURSOR

El mensaje WM_SETCURSOR nos permite un control mucho mayor sobre el aspecto del cursor, incluso aunque éste abandone el área de cliente. Se recibe cada vez que el cursor se mueve dentro de la ventana, y de ese modo podemos cambiar la apariencia del cursor a capricho.
Podemos, por ejemplo, cambiar el cursor dependiendo de la zona de la ventana, verificando si se encuentra dentro de un rectángulo o de una región determinada.
Si el cursor está fuera de cualquiera de las zonas de nuestro interés, probablemente queramos que su aspecto sea el esperado, por ejemplo, cuando esté sobre un borde o sobre la barra de menús. En este caso, debemos dejar que el mensaje se procese por el procedimiento por defecto:

```
        case WM_SETCURSOR:
           SetRect(&re, 20, 20, 80,80);
           GetCursorPos(&punto);
           ScreenToClient(hwnd, &punto);
           if(PtInRect(&re, punto))

SetCursor(LoadCursorFromFile("horse.ani"));
           else
              return    DefWindowProc(hwnd,    msg,
wParam, lParam);
           break;
```

En este ejemplo, si el cursor está sobre el rectángulo de coordenadas (20,20)-(80,80) se mostrará el cursor del caballo, en caso contrario, se ejecuta el proceso del mensaje por defecto, es decir: si el cursor está sobre el área de cliente, se mostrará el cursor de la clase, y en el área de no cliente, se mostrará el cursor que corresponda.
El problema con los cursores animados es que cada vez que se recibe el mensaje WM_SETCURSOR se vuelve a asignar el cursor, y éste vuelve a la primera imagen de la animación. Deberíamos crear un procedimiento que sólo lo cambie la primera vez que entramos en el área especificada, y no cada vez que se procese el mensaje, esto no sucede si cambiamos el cursor de clase.

Ocultar y mostrar

También podemos ocultar o mostrar el cursor en cualquier momento, para ello usaremos la función ShowCursor. Esta función admite un parámetro de tipo *BOOL*, si es TRUE, el valor del contador se incrementa, si es FALSE, se decrementa. Si el valor del contador es mayor o igual que cero, el cursor se muestra, en caso contrario, se oculta.
La utilidad de ocultar el cursor es limitada, tal vez, evitar que el usuario lleve a cabo ciertas tareas que impliquen el uso de ratón en determinados momentos.

Confinar el cursor

En ocasiones nos puede interesar que el cursor no salga de cierta zona de la pantalla hasta que se se cumplan ciertas condiciones especiales, por ejemplo, podemos confinar el cursor a una zona concreta de una ventana, y limitar de este modo las posibilidades de mover el cursor, o de hacer clic en ciertas partes de la pantalla.

Para esto disponemos de la función ClipCursor, que admite un parámetro de tipo RECT que define el rectángulo del que el cursor no puede salir. La función GetClipCursor permite recuperar ese rectángulo.

El cursor está asociado al ratón, y ambos son recursos únicos en el sistema, es decir, una aplicación no debería adueñarse de ellos de forma exclusiva. Cuando una aplicación que ha confinado el cursor pierde el foco, debe liberarlo para que pueda acceder a cualquier punto de la pantalla.

Esto se puede hacer procesando los mensajes WM_SETFOCUS y WM_KILLFOCUS:

```
        case WM_KILLFOCUS:
           ClipCursor(NULL);
           break;
        case WM_SETFOCUS:
           GetWindowRect(hwnd, &re);
           ClipCursor(&re);
           break;
```

Este ejemplo confina el cursor a la ventana de la aplicación, e impide que el cursor se use fuera de ella. Si la aplicación pierde el foco, el cursor se libera, y cuando lo recupera, se vuelve a confinar.

Pero no bastará con esto, ya que cada vez que la ventana se mueva o cambie de tamaño, el cursor vuelve a quedar libre. Para evitar que esto suceda podemos recurrir a dos nuevos mensajes: WM_SIZE y WM_MOVE, que se reciben cuando la ventana cambia de tamaño o de posición, respectivamente.

Sencillamente, procesaremos los mensajes WM_SETFOCUS, WM_SIZE y WM_MOVE del mismo modo:

```
        case WM_KILLFOCUS:
           ClipCursor(NULL);
           break;
        case WM_MOVE:
        case WM_SIZE:
        case WM_SETFOCUS:
           GetWindowRect(hwnd, &re);
           ClipCursor(&re);
```

```
            break;
```

Destrucción de cursores

Cursores creados con CreateIconIndirect se destruyen con DestroyCursor, no es necesario destruir el resto de los cursores.

Capítulo 33 El ratón

Aunque importante, se considera que el ratón no es imprescindible en Windows, por lo tanto, debemos incluir todo lo necesario para que nuestras aplicaciones se puedan manejar exclusivamente con el teclado. Esta es la recomendación de Windows, sin embargo, no todo el mundo la sigue, y a menudo (cada vez más) encontramos aplicaciones que no pueden manejarse sin ratón.

Todas las entradas procedentes del ratón se reciben mediante mensajes. Así que si nuestra aplicación quiere procesar el ratón como una entrada, debe procesar esos mensajes.

Como vimos en el capítulo anterior, el ratón está asociado al cursor, cuando el primero se mueve, el segundo se desplaza en pantalla para indicar dicho movimiento. Tenemos esto tan asumido que frecuentemente decimos que movemos tanto el cursor como el ratón indistintamente. La ventana que recibe los mensajes del ratón es sobre la que se sitúa el punto activo del cursor (hotspot).

Capturar el ratón

Como si fuesemos un gato, podemos capturar el ratón y mantenerlo cautivo para nuestra aplicación usando la función SetCapture e indicando qué ventana es la que captura el ratón. Para liberarlo se puede usar la función ReleaseCapture, pero también se liberará si otra ventana captura el ratón o si el usuario hace clic en otra ventana distinta de la que lo ha capturado.

Cada vez que el ratón es capturado, se envía el mensaje WM_CAPTURECHANGED a la ventana que pierde la captura.

Un caso típico de captura de ratón es el del arrastre de objetos de una ventana a otra, por ejemplo, pulsamos el botón izquierdo del ratón sobre el icono correspondiente a un fichero, y manteniéndolo pulsado movemos el cursor a otra ventana, sólo entonces soltamos el botón. Si queremos que la primera ventana siga recibiendo los mensajes del ratón aunque el cursor salga de sus límites, incluido el mensaje de soltar el botón, deberemos capturar el ratón.

No todos los mensajes sobre eventos del ratón son enviados a la ventana que lo ha capturado. Por ejemplo si el cursor se desplaza sobre ventanas diferentes a la que ha capturado el ratón, los mensajes sobre el movimiento del cursor se envían a esas ventanas, salvo que uno de los botones del ratón permanezca pulsado.

Otro efecto secundario destacable es que también perderemos las funciones normales del ratón sobre las ventanas hijas de la que ha capturado el ratón. Es decir, si capturamos el ratón, los clics sobre controles o menús de la ventana no

realizan sus acciones habituales. De modo que no podremos acceder al menú, ni activar controles mediante el ratón.

Configuración

Para saber si el ratón está presente se puede usar la función GetSystemMetrics, con el parámetro SM_MOUSEPRESENT. Además, podemos averiguar el número de botones del ratón, usando la misma función, con el parámetro SM_CMOUSEBUTTONS. Se puede trabajar con ratones de uno, dos o tres botones. Los llamaremos izquierdo, derecho y central, y frecuentemente apareceran con sus inicales en inglés: L, R y M, respectivamente.
Las funciones de los botones izquierdo y derecho se pueden intercambiar cuando el usuario lo maneja con la mano izquierda (o si quiere hacerlo), mediante la función SwapMouseButton. Esto significa que el botón izquierdo genera los mensajes del botón derecho, y viceversa.
Hay que tener en cuenta que el ratón es un recurso compartido, por lo tanto, esta modificación afectará a todas las ventanas.

Mensajes

Cuando ocurre un evento relacionado con el ratón: pulsaciones de botones o movimientos, se envía un mensaje, y junto con él, las coordenadas del punto activo del cursor. Además, las ventanas siguen recibiendo estos mensajes aunque no tengan el foco del teclado. También los recibirán si la ventana ha capturado el ratón, aunque el cursor no esté sobre la ventana.
Los mensajes del ratón se envían del modo "post", es decir, son mensajes "lentos". En realidad se colocan en una cola que se procesa cuando el sistema tiene tiempo libre. Si se generan muchos mensajes en poco tiempo, el sistema elimina automáticamente los más antiguos, de modo que la aplicación sólo recibe los últimos.
Los mensajes "rápidos" se envían en modo "send", y el control pasa directamente al procedimiento de ventana, es decir, todos los mensajes de este tipo se procesan.

Nota: lo siento, pero no he encontrado traducción para los términos "send" y "post" que indiquen todos los matices implícitos en inglés. Podríamos decir que los mensajes enviados "send" viajan por teléfono, el receptor los recibe tan pronto se generan, los del tipo "post" viajan por correo, y es posible que los últimos mensajes invaliden los anteriores o sencillamente, los hagan inútiles.

Mensajes del área de cliente

Normalmente sólo procesaremos estos mensajes, e ignoraremos el resto.
El mensaje WM_MOUSEMOVE se recibe cada vez que el usuario mueve el cursor sobre la ventana. Este mensaje es el que más frecuentemente se elimina, ya que puede ser generado muchas veces por segundo.
Existen otros mensajes, uno por cada evento de pulsar o soltar un botón, o por un doble clic, y para cada uno de estos tres eventos, variantes para cada uno de los tres botones del ratón. En total nueve mensajes.

WM_LBUTTONDOWN, WM_MBUTTONDOWN y WM_RBUTTONDOWN se envían cada vez que el usuario pulsa el botón izquierdo, central o derecho, respectivamente.
WM_LBUTTONUP, WM_MBUTTONUP y WM_RBUTTONUP se envían cuando el usuario suelta cada uno de los botones izquierdo, central o derecho, respectivamente.
WM_LBUTTONDBLCLK, WM_MBUTTONDBLCLK y WM_RBUTTONDBLCLK se envían cuando el usuario hace doble clic sobre el botón izquierdo, central o derecho respectivamente. En estos casos también se envían los mensajes correspondientes a las pulsaciones y sueltas individuales que completan el doble clic. Por ejemplo, un mensaje WM_LBUTTONDBLCLK, se recibe dentro de una secuencia de mensajes WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK y WM_LBUTTONUP.
Para que se genere un mensaje de doble clic se deben cumplir ciertos requisitos. El segundo clic debe producirse en un área determinada alrededor del primero, y dentro de un intervalo de tiempo determinado. Tanto las dimensiones del área (en realidad un rectángulo), como el tiempo de doble clic se pueden modificar mediante funciones del API, pero es mejor dejar este trabajo al usuario mediante Panel de Control, ya que estos cambios afectan a todas las ventanas.
Las ventanas no reciben los mensajes de doble clic por defecto, hay que activar un estilo determinado para que esto sea así. En concreto, al ventana debe crearse a partir de una clase que tenga el estilo CS_DBLCLKS. Hasta ahora siempre hemos creado ventanas de este tipo en nuestros ejemplos:

```
    WNDCLASSEX wincl;         /* Estructura de datos
para la clase de ventana */

    /* Estructura de la ventana */
    wincl.hInstance = hThisInstance;
    wincl.lpszClassName = "NUESTRA_CLASE";
    wincl.lpfnWndProc  =  WindowProcedure;          /*
Esta función es invocada por Windows */
    wincl.style = CS_DBLCLKS;                       /*
Captura los doble-clicks */
    wincl.cbSize = sizeof (WNDCLASSEX);

    /* Usar icono y puntero por defector */
    wincl.hIcon = LoadIcon (hThisInstance, "Icono");
    wincl.hIconSm    =    LoadIcon    (hThisInstance,
"Icono");
    wincl.hCursor = NULL;
    wincl.lpszMenuName = "Menu";
    wincl.cbClsExtra  =  0;       /*  Sin  información
adicional para la */
```

```
    wincl.cbWndExtra = 0;     /* clase o la ventana
*/
    /* Usar el color de fondo por defecto para es
escritorio */
    wincl.hbrBackground                              =
GetSysColorBrush(COLOR_BACKGROUND);

    /* Registrar la clase de ventana, si falla,
salir del programa */
    if(!RegisterClassEx(&wincl)) return 0;
```

En todos los casos, en el parámetro lParam de cada mensaje se reciben las coordenadas del punto activo del cursor. En la palabra de menor peso la coordenada x y en la de mayor peso, la coordenada y. Además, las coordenadas son coordenadas de cliente, es decir, relativas a la esquina superior izquierda del área de cliente. Para separar estos valores se puede usar la macro MAKEPOINTS, que convierte el valor en el parámetro lParam en una estructura POINTS.
También en todos los casos, el parámetro wParam del mensaje contiene información sobre si ciertas teclas o botones del ratón están pulsados.

Valor	Descripción
MK_CONTROL	Activo si la tecla CTRL está pulsada.
MK_LBUTTON	Activo si el botón izquierdo del ratón está pulsado.
MK_MBUTTON	Activo si el botón central del ratón está pulsado.
MK_RBUTTON	Activo si el botón derecho del ratón está pulsado.
MK_SHIFT	Activo si la tecla MAYÚSCULAS está pulsada.

```
    static int izq, cen, der;
    static POINTS punto;
...
        case WM_MOUSEMOVE:
           punto = MAKEPOINTS(lParam);
           izq = (wParam & MK_LBUTTON) ? 1 : 0;
           cen = (wParam & MK_MBUTTON) ? 1 : 0;
           der = (wParam & MK_RBUTTON) ? 1 : 0;
           hdc = GetDC(hwnd);
           Pintar(hdc, izq, cen, der, punto, TRUE);
           ReleaseDC(hwnd, hdc);
           break;
        case WM_LBUTTONDOWN:
```

```
            izq = 2;
            punto = MAKEPOINTS(lParam);
            hdc = GetDC(hwnd);
            Pintar(hdc, izq, cen, der, punto, TRUE);
            ReleaseDC(hwnd, hdc);
            break;
        case WM_LBUTTONUP:
            izq = 3;
            punto = MAKEPOINTS(lParam);
            hdc = GetDC(hwnd);
            Pintar(hdc, izq, cen, der, punto, TRUE);
            ReleaseDC(hwnd, hdc);
            break;
        case WM_LBUTTONDBLCLK:
            izq = 4;
            punto = MAKEPOINTS(lParam);
            hdc = GetDC(hwnd);
            Pintar(hdc, izq, cen, der, punto, TRUE);
            ReleaseDC(hwnd, hdc);
            break;
```

Mensajes del área de no cliente

Existe la misma gama de mensajes de ratón para el área de no cliente que para el área de cliente, el nombre de los mensajes es el mismo, pero con la letras NC después del '_', por ejemplo, el mensaje que notifica movimientos del cursor dentro del área de no cliente es WM_NCMOUSEMOVE.

Del mismo modo, los mensajes relacionados con clics de botones son:

WM_NCLBUTTONDOWN, WM_NCMBUTTONDOWN y WM_NCRBUTTONDOWN se envían cada vez que el usuario pulsa el botón izquierdo, central o derecho, respectivamente.

WM_NCLBUTTONUP, WM_NCMBUTTONUP y WM_NCRBUTTONUP se envían cuando el usuario suelta cada uno de los botones izquierdo, central o derecho, respectivamente.

WM_NCLBUTTONDBLCLK, WM_NCMBUTTONDBLCLK y WM_NCRBUTTONDBLCLK se envían cuando el usuario hace doble clic sobre el botón izquierdo, central o derecho respectivamente. En estos casos también se envían los mensajes correspondientes a las pulsaciones y sueltas individuales que completan el doble clic. Por ejemplo, un mensaje WM_NCLBUTTONDBLCLK, se recibe dentro de una secuencia de mensajes WM_NCLBUTTONDOWN, WM_NCLBUTTONUP, WM_NCLBUTTONDBLCLK y WM_NCLBUTTONUP.

Estos mensajes se deben procesar con cuidado, ya que al hacer que la aplicación responda a ellos podremos perder muchas de las funciones propias del área de no cliente, como acceso a menús, movimiento de la ventana, cambio de tamaño, etc,

salvo que llamemos al procedimiento de ventana por defecto después de procesar el mensaje. Pero generalmente no tendremos necesidad de procesar estos mensajes.
En el caso de estos mensajes, el parámetro *wParam* no contiene información sobre el estado de teclas y botones, sino sobre el *hit test*, la zona en la que se encuentra el punto activo del cursor. De este modo podemos saber si está sobre un borde, un menú, la barra de título, etc. Veremos esto en más detalle más abajo, al ver el mensaje WM_NCHITTEST.
El parámetros *lParam* sighe conteniendo las coordenadas del cursor, pero en coordenadas de pantalla.

```
        case WM_NCMOUSEMOVE:
           punto = MAKEPOINTS(lParam);
           izq = cen = der = 0;
           hdc = GetDC(hwnd);
           Pintar(hdc, izq, cen, der, punto, FALSE);
           ReleaseDC(hwnd, hdc);
           break;
        case WM_NCLBUTTONDOWN:
           izq = 2;
           punto = MAKEPOINTS(lParam);
           hdc = GetDC(hwnd);
           Pintar(hdc, izq, cen, der, punto, FALSE);
           ReleaseDC(hwnd, hdc);
           return  DefWindowProc(hwnd,  msg,  wParam,
lParam);
           break;
        case WM_NCLBUTTONUP:
           izq = 3;
           punto = MAKEPOINTS(lParam);
           hdc = GetDC(hwnd);
           Pintar(hdc, izq, cen, der, punto, FALSE);
           ReleaseDC(hwnd, hdc);
           return  DefWindowProc(hwnd,  msg,  wParam,
lParam);
           break;
        case WM_NCLBUTTONDBLCLK:
           izq = 4;
           punto = MAKEPOINTS(lParam);
           hdc = GetDC(hwnd);
           Pintar(hdc, izq, cen, der, punto, FALSE);
           ReleaseDC(hwnd, hdc);
```

```
            return  DefWindowProc(hwnd,  msg,  wParam,
lParam);
            break;
```

Mensaje WM_NCHITTEST

El mensaje WM_NCHITTEST se envía a la ventana que contiene el cursor, o a la que ha capturado el ratón, cada vez que ocurre un evento del ratón. Este mensaje se usa por Windows para determinar si se debe enviar un nuevo mensaje de área de cliente o de área de no cliente. Si queremos que nuestra aplicación reciba mensajes del ratón debemos dejar que la función DefWindowProc procese este mensaje.

```
    static POINTS puntoHT;
    static LRESULT hittest;
...
        case WM_NCHITTEST:
           puntoHT = MAKEPOINTS(lParam);
           hdc = GetDC(hwnd);
           sprintf(cad,   "Punto   HIT-TEST:   (%4d,
%4d)",
              puntoHT.x, puntoHT.y);
           TextOut(hdc, 10, 110, cad, strlen(cad));
           MostrarHitTest(hdc, hittest);
           ReleaseDC(hwnd, hdc);
           hittest   =   DefWindowProc(hwnd,   msg,
wParam, lParam);
           return hittest;
```

En el parámetro *lParam* se reciben las coordenadas del punto activo del cursor en coordenadas de pantalla:

Cuando la función DefWindowProc procesa este mensaje devuelve un código de hit-test, que depende de la zona en que se encuentre el cursor.

Valor	Posición del punto activo
HTBORDER	En el borde de la ventana que no tiene borde de cambio de tamaño.
HTBOTTOM	En borde horizontal inferior de una ventana.
HTBOTTOMLEFT	En la esquina inferior izquierda del borde de una ventana.
HTBOTTOMRIGHT	En la esquina inferior derecha del borde de una ventana.
HTCAPTION	En una barra de título.

HTCLIENT	En un área de cliente.
HTERROR	En el fondo de la pantalla o en una línea de división entre ventanas (lo mismo que HTNOWHERE, excepto que DefWindowProc produce un pitido de sistema para indicar un error).
HTGROWBOX	En una caja de cambio de tamaño (lo mismo que HTSIZE).
HTHSCROLL	En la barra de desplazamiento horizontal.
HTLEFT	En el borde izquierdo de una ventana.
HTMENU	En un menú.
HTNOWHERE	En el fondo de la pantalla o en una línea de división entre ventanas.
HTREDUCE	En un botón de minimizar.
HTRIGHT	En el borde derecho de una ventana.
HTSIZE	En una caja de cambio de tamaño (lo mismo que HTGROWBOX).
HTSYSMENU	En un menú de sistema o en un botón de cierre en una ventana hija.
HTTOP	En el borde horizontal superior de una ventana.
HTTOPLEFT	En la esquina superior izquierda del borde de una ventana.
HTTOPRIGHT	En la esquina superior derecha de un borde de ventana.
HTTRANSPARENT	En una ventana actualmente tapada por otra ventana.
HTVSCROLL	En la barra de desplazamiento vertical.
HTZOOM	En un botón de maximizar.

Si el valor devuelto por el procedimiento de ventana es HTCLIENT es porque está en el área de cliente, en ese caso las coordenadas se trasladan a coordenadas de cliente y se envía un mensaje "lento "de área de cliente, y en el parámetro *wParam* se envía el estado de los botones del ratón. Si se encuentra en otra zona, se envía un mensaje "lento" de área de no cliente, se mantienen las coordenadas en coordenadas de pantalla y en el parámetro *wParam* se envía el código hit-test.

Mensaje WM_MOUSEACTIVATE

El mensaje WM_MOUSEACTIVATE se envía a una ventana cuando se hace clic con uno de los botones del ratón cuando el cursor está sobre ella o sobre una de sus ventanas hijas, y si la ventana está inactiva. Este mensaje se envía después del mensaje WM_NCHITTEST y antes de cualquier mensaje de área de cliente o de área de ni cliente.

Si se deja que DefWindowProc procese este mensaje, se activará la ventana y se enviará el mensaje de pulsación de botón a la ventana.
Si procesamos el mensaje en nuestro procedimiento de ventana tenemos más opciones, y podemos controlar si se activa o no la ventana y si se descarta el mensaje de pulsación del botón o no. Podemos devolver los siguientes valores para conseguir estos resultados al procesar este mensaje:

Valor	Significado
MA_ACTIVATE	Activa la ventana, y no descarta el mensaje de ratón.
MA_ACTIVATEANDEAT	Activa la ventana, y descarta el mensaje de ratón.
MA_NOACTIVATE	No activa la ventana, y no descarta el mensaje de ratón.
MA_NOACTIVATEANDEAT	No activa la ventana, pero descarta el mensaje de ratón.

Otros mensajes de ratón

Algunas de las características que ahora son corrientes en el manejo del ratón, no lo eran o ni siquiera existían hace unos años. Por ejemplo, la rueda del ratón es un invento relativamente reciente. Además, debido a los navegadores de Internet, se han popularizado algunos eventos relacionados con el ratón que antes no existían, como el paso sobre zonas determinadas, o la transición entre unas zonas y otras de la pantalla. Estos eventos se usan en los navegadores para cambiar el aspecto de textos o botones, y de ese modo llamar la atención del usuario sobre ellos para indicar que son zonas activas a clics.
Veamos ahora estos nuevos mensajes en el API.

Mensaje WM_MOUSEWHEEL (Windows NT)

El mensaje WM_MOUSEWHEEL se envía cada vez que el usuario activa la rueda de desplazamiento del ratón. Los parámetros de este mensaje son los mismos que en el mensaje WM_MOUSEMOVE, pero para incluir la información sobre el avance o retroceso de la rueda, el parámetro *wParam* se ha dividido en dos. En la parte baja se empaquetan las banderas sobre el estado de los botones, y en la parte alta el valor de avance o retroceso de la rueda. Usaremos las macros LOWORD y HIWORD para extraer esos valores.
Los valores de avance y retroceso de la rueda serán siempre múltiplos de 120. Esto se ha hecho así para permitir que en el futuro se puedan construir dispositivos compatibles con la rueda, pero que proporcionen mayor precisión. Debemos considerar siempre que una unidad de desplazamiento de la rueda es 120, o para evitar problemas de dependencias, de la constante **WHEEL_DELTA**.
Los valores positivos indican movimientos de rueda hacia adelante, y los negativos, hacia atrás.

```
        case WM_MOUSEWHEEL:
           punto = MAKEPOINTS(lParam);
```

```
            izq = (LOWORD(wParam) & MK_LBUTTON) ? 1 :
0;
            cen = (LOWORD(wParam) & MK_MBUTTON) ? 1 :
0;
            der = (LOWORD(wParam) & MK_RBUTTON) ? 1 :
0;
            rueda = HIWORD(wParam);
            rueda /= WHEEL_DELTA;
            hdc = GetDC(hwnd);
            Pintar(hdc, izq, cen, der, punto, TRUE);
            sprintf(cad, "rueda = %04d ", rueda);
            TextOut(hdc, 10, 130, cad, strlen(cad));
            ReleaseDC(hwnd, hdc);
            break;
```

Trazar eventos del ratón (Windows NT)

Nuestra aplicación puede recibir dos mensajes sobre eventos del ratón: WM_MOUSELEAVE y WM_MOUSEHOVER, cuando el ratón abandona una ventana o cuando permanece sobre un área determinada de la ventana durante un periodo de tiempo, respectivamente. Pero para que esto suceda, previamente debemos activar el trazado de eventos, mediante una llamada a la función TrackMouseEvent.

Esta función necesita como parámetro un puntero a una estructura TRACKMOUSEEVENT. En esa estructura indicamos qué eventos queremos que se notifique, y el tiempo necesario para generar un mensaje WM_MOUSEHOVER:

```
        case WM_NCHITTEST:
            hittest    =    DefWindowProc(hwnd,    msg,
wParam,  lParam);
            if(HTCLIENT == hittest) {
               if(!dentro) {
                  dentro = TRUE;
                  tme.cbSize                          =
sizeof(TRACKMOUSEEVENT);
                  tme.dwFlags    =    TME_HOVER    |
TME_LEAVE;
                  tme.hwndTrack = hwnd;
                  tme.dwHoverTime = 1000;
                  TrackMouseEvent(&tme);
               }
```

```
        }
        return hittest;
        break;
```

En este ejemplo llamamos a TrackMouseEvent para que nos notifique ambos eventos para la ventana *hwnd*, y ajustamos el tiempo *Hover* en un segundo.

Mensaje WM_MOUSELEAVE (Windows NT)

Si hemos activado el evento *Leave* recibiremos un mensaje WM_MOUSELEAVE cuando el cursor abandone el área de cliente de la ventana. Una vez que se recibe el mensaje no se vuelve a generar, salvo que volvamos a activar el evento, usando de nuevo la función TrackMouseEvent.

```
    case WM_MOUSELEAVE:
        dentro = FALSE;
        hdc = GetDC(hwnd);
        TextOut(hdc, 10, 170, "Leave", 5);
        ReleaseDC(hwnd, hdc);
        break;
```

Mensaje WM_MOUSEHOVER (Windows NT)

Del mismo modo, si hemos activado el evento *Hover* recibiremos un mensaje WM_MOUSEHOVER cuando haya transcurrido el tiempo indicado y el cursor no se haya movido de una zona determinada del área de cliente de la ventana. Esta zona está predederminada por Windows, aunque se puede modificar su tamaño (ver TRACKMOUSEEVENT. Una vez que se recibe el mensaje no se vuelve a generar, salvo que volvamos a activar el evento, usando de nuevo la función TrackMouseEvent.

```
    case WM_MOUSEHOVER:
        hdc = GetDC(hwnd);
        TextOut(hdc, 10, 170, "Hover", 5);
        ReleaseDC(hwnd, hdc);
        break;
```

Arrastrar objetos

Una de las operaciones más frecuentes que se realizan mediante el ratón es la de arrastrar objetos. No entraremos en muchos detalles por ahora, Windows dispone de formas especiales de realizar arrastre de objetos entre distintas ventanas y aplicaciones, pero de momento veremos un ejemplo sencillo sobre cómo arrastrar objetos dentro de una misma ventana.

En este ejemplo usaremos iconos como objetos. A cada icono le corresponde una imagen y una posición en pantalla:

```
typedef struct {
    HICON icono;
    POINT coordenada;
} Objeto;
```

Al procesar el mensaje WM_CREATE inicializamos el array de objetos:

```
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           objeto[0].icono   =   LoadIcon(hInstance,
"ufo");
           objeto[0].coordenada.x = 10;
           objeto[0].coordenada.y = 10;
           objeto[1].icono   =   LoadIcon(hInstance,
"libro");
           objeto[1].coordenada.x = 10;
           objeto[1].coordenada.y = 50;
           objeto[2].icono   =   LoadIcon(hInstance,
"mundo");
           objeto[2].coordenada.x = 10;
           objeto[2].coordenada.y = 90;
           objeto[3].icono   =   LoadIcon(hInstance,
"hamburguesa");
           objeto[3].coordenada.x = 50;
           objeto[3].coordenada.y = 10;
           objeto[4].icono   =   LoadIcon(hInstance,
"smile");
           objeto[4].coordenada.x = 50;
           objeto[4].coordenada.y = 50;
           capturado = -1;
           break;
```

Una operación de arrastre comienza con un clic sobre un objeto. Después, sin soltar el botón del ratón, se desplaza el ratón, y con él el obejto, a la posición deseada, y finalmente, se suelta el botón del ratón en esa posición.
Lo primero es seleccionar el objeto a arrastrar. Para ello procesaremos el mensaje WM_LBUTTONDOWN, y comprobaremos si las coordenadas del cursor corresponden con alguno de los objetos que es posible arrastrar. Si es así, guardamos en una variable el identificador del objeto, y al mismo tiempo, activamos el modo de arrastre:

```
        case WM_LBUTTONDOWN:
           punto = MAKEPOINTS(lParam);
           for(i = 0; capturado == -1 && i < 5; i++)
{
              SetRect(&re,
                objeto[i].coordenada.x,
                objeto[i].coordenada.y,
                objeto[i].coordenada.x+32,
                objeto[i].coordenada.y+32);
              POINTSTOPOINT(lpunto, punto);
              if(PtInRect(&re, lpunto)) {
                 capturado = i;
                 ClientToScreen(hwnd,
&objeto[i].coordenada);

SetCursorPos(objeto[i].coordenada.x,
                    objeto[i].coordenada.y);
                 SetCapture(hwnd);
                 ShowCursor(FALSE);
              }
           }
           break;
```

Si la variable *capturado* vale -1, indica que no se está realizando un arrastre, si tiene otro valor, indica el objeto arrestrado.
Para cada objeto calculamos las coordenadas de un rectángulo que lo contiene, y comprobamos si la posición actual del cursor está dentro del rectángulo. Si es así, actualizamos el valor de *capturado*, cambiamos la posición del cursor a la esquina superior izquierda del objeto, y ocultamos el cursor.
El cambio de coordenadas sirve para eliminar el salto que se produciría cuando pinchemos sobre un punto distinto de la esquina superior izquierda. Ocultar el cursor hace que el objeto arrastrado parezca sustituir al cursor, y hace más fácil arrastrarlo con precisión.
Además, capturamos el ratón para que la operación de arrastre no se interrumpa si el cursor sale del área de cliente de la ventana.
Durante el arrastre debemos procesar el mensaje WM_MOUSEMOVE. Cada vez que recibamos el mensaje, borraremos el objeto en su posición actual, actualizamos las coordenadas según la posición del cursor, y volvemos a dibujarlo en la nueva posición:

```
        case WM_MOUSEMOVE:
           punto = MAKEPOINTS(lParam);
```

```
            if(capturado != -1) {
               hdc = GetDC(hwnd);
               //drag
               // Borrar en posición actual:
               SetRect(&re,
                  objeto[capturado].coordenada.x,
                  objeto[capturado].coordenada.y,
                  objeto[capturado].coordenada.x+32,
                  objeto[capturado].coordenada.y+32);
               FillRect(hdc,                        &re,
GetSysColorBrush(COLOR_BACKGROUND));
               // Actualizar coordenadas:

POINTSTOPOINT(objeto[capturado].coordenada, punto);
               // Pintar en nueva posición:
               DrawIcon(hdc,
                  objeto[capturado].coordenada.x,
                  objeto[capturado].coordenada.y,
                  objeto[capturado].icono);
               ReleaseDC(hwnd, hdc);
               InvalidateRect(hwnd, &re, FALSE);
            }
            break;
```

Para borrar pintamos usando el color de fondo, la zona ocupara por el objeto. Para actualizar la posición del objeto usamos la macro POINTSTOPOINT, esto es porque la posición del cursor se almacena en una estructura POINTS, y la del objeto en una estructura POINT. A continuación mostramos el objeto, e invalidamos la zona que ocupaba originalmente. Esto último es necesario, ya que el objeto arrastrado puede pasar sobre otros objetos borrándolos.

Finalmente, cuando soltemos el botón se recibirá un mensaje WM_LBUTTONUP. En ese momento debemos dar por terminada la operación de arrastre, y regresaremos al estado inicial:

```
         case WM_LBUTTONUP:
            if(capturado != -1) {
               capturado = -1;
               InvalidateRect(hwnd, NULL, FALSE);
               ReleaseCapture();
               ShowCursor(TRUE);
            }
```

```
            break;
```

Asignamos -1 a *capturado*, redibujamos toda la ventana, libreramos el ratón y mostramos el cursor.

Capítulo 34 El Teclado

Al igual que el ratón, las entradas del teclado se reciben en forma de mensajes. En este capítulo veremos el manejo básico del teclado, y algunas características relacionadas con este dispositivo.

Como pasa con otros dispositivos del ordenador, en el teclado distinguimos al menos dos capas: la del dispositivo físico y la del dispositivo lógico.

En cuanto al dispositivo físico, el teclado no es más que un conjunto de teclas. Cada una de ellas genera un código diferente, cada vez que es pulsada o liberada, a esos códigos los llamaremos códigos de escaneo (*scan codes*). Por supuesto, dado que estamos en la capa física, estos códigos son dependientes del dispositivo, y en principio, cambiarán dependiendo del fabricante del teclado.

Pero Windows nos permite hacer nuestros programas independientes del dispositivo, de modo que no será frecuente que tengamos que trabajar con códigos de escaneo, y aunque el API nos informe de esos códigos, generalmente los ignoraremos.

En la capa lógica, el driver del teclado traduce los códigos de escaneo a códigos de tecla virtual (*virtual-key codes*). Estos códigos son independientes del dispositivo, e identifican el propósito de cada tecla. Generalmente, serán esos los códigos que usemos en nuestros programas. (Tabla al final).

El Foco del teclado

Los mensajes del teclado se envían al proceso de primer plano que haya creado la ventana que actualmente tiene el foco del teclado. El teclado se comparte entre todas las ventanas abiertas, pero sólo una de ellas puede poseer el foco del teclado, los mensajes del teclado llegan a esa ventana a través del bucle de mensajes del proceso que las creó.

Sólo una ventana, o ninguna, puede poseer el foco del teclado, para averiguar cual es la que lo posee podemos usar la función GetFocus, siempre que esa ventana pertenezca al proceso actual. Para asignar el foco a la ventana que queramos se usa la función SetFocus. Ya hemos usado esta función anteriormente, para cambiar el control que recibe la entrada del usuario al iniciar un cuadro de diálogo.

Cuando una ventana pierde el foco, recibe un mensaje WM_KILLFOCUS y a la ventana que lo recibe, se le envía un mensaje WM_SETFOCUS. En ocasiones se puede usar el mensaie WM_KILLFOCUS para realizar la validación de los datos de un control, o cualquier tarea, antes de perder definitivamente la atención del usuario. Del mismo modo, el mensaje WM_SETFOCUS se puede usar para preparar una ventana o control antes de que el usuario pueda modificar los datos que contiene. Normalmente, si una ventana acepta entradas desde el teclado, sabremos si tiene el foco porque se muestra un caret en su interior.

Una propiedad relacionada con el foco del teclado es el de ventana activa. La ventana activa es con la que el usuario está trabajando. La ventana con el foco del teclado es, o bien la ventana activa, o bien una de sus ventanas hija. La ventana activa se distingue porque su barra de título cambia de color, y porque suele tener el foco del teclado y del ratón (aunque esto no siempre es cierto).
El usuario puede cambiar de ventana activa, usando el teclado, haciendo clic sobre ella, etc. También es posible hacerlo usando la función SetActiveWindow y con GetActiveWindow, un proceso puede obtener el manipulador de la ventana activa asociada, si es que existe. Cada vez que una ventana deja de ser la activa, recibe un mensaje WM_ACTIVATE, y después se envía el mismo mensaje a la que pasa a ser activa.

```
        case WM_ACTIVATE:
           if((HWND)lParam  ==  NULL)  strcpy(nombre,
"NULL");
           else  GetWindowText((HWND)lParam,  nombre,
128);
           if(LOWORD(wParam) == WA_INACTIVE)
              sprintf(cad,    "Ventana    desactivada
hacia %s", nombre);
           else
              sprintf(cad,  "Ventana  activada  desde
%s", nombre);
           break;
        case WM_KILLFOCUS:
           if((HWND)wParam  ==  NULL)  strcpy(nombre,
"NULL");
           else  GetWindowText((HWND)wParam,  nombre,
128);
           sprintf(cad, "Pérdida de foco en favor de
%s", nombre);
           break;
        case WM_SETFOCUS:
           if((HWND)wParam  ==  NULL)  strcpy(nombre,
"NULL");
           else  GetWindowText((HWND)wParam,  nombre,
128);
           sprintf(cad, "Foco recuperado desde %s",
nombre);
           break;
```

Ventanas inhibidas

A veces es útil hacer que una ventana no pueda recibir el foco del teclado, ya sea porque los datos que contiene no deban ser modificados, o porque no tengan sentido en un contexto determinado. En ese caso, podemos inhibir tal ventana usando la función EnableWindow. La misma función se usa para desinhibirla. Una ventana inhibida no puede recibir mensajes del teclado ni del ratón.

```
    static BOOL cambio;
...
          cambio = FALSE;
          EnableWindow(GetDlgItem(hDlg,
ID_CONTROL1), !cambio);
          EnableWindow(GetDlgItem(hDlg,
ID_CONTROL2), cambio);
          SetFocus(GetDlgItem(hDlg, ID_CONTROL1));
```

Mensajes de pulsación de teclas

La acción de pulsar una tecla implica dos eventos, uno cuando se pulsa y otro cuando se libera. Cuando se pulsa una tecla se envía un mensaje WM_KEYDOWN o WM_SYSKEYDOWN a la ventana que tiene el foco del teclado, y cuando se libera, un mensaje WM_KEYUP o WM_SYSKEYUP.
Los mensajes WM_SYSKEYDOWN y WM_SYSKEYUP se refieren a teclas de sistema. Las teclas de sistema son las que se pulsan manteniendo pulsada la tecla [Alt]. Los otros dos mensajes se refieren a teclas que no sean de sistema.
En todos los casos, el parámetro *wParam* contiene el código de tecla virtual, y el parámetro *lParam* varios datos asociados a la tecla, como repeticiones, código de escaneo, si se trata de una tecla extendida, el código de contexto, el estado previo de la tecla y el estado de transición.
Podemos crear un campo de bits para tratar estos datos más fácilmente:

```
typedef union {
   struct {
      unsigned int repeticion:16;
      unsigned int scan:8;
      unsigned int extendida:1;
      unsigned int reservado:4;
      unsigned int contexto:1;
      unsigned int previo:1;
      unsigned int transicion:1;
   };
   unsigned int lParam;
} keyData;
```

Cuando el usuario deja pulsada una tecla generalmente tiene la intención de repetir varias veces esa pulsación. El sistema está preparado para ello, y a partir de cierto momento, se generará una repetición cada cierto intervalo de tiempo. Los dos tiempos se pueden ajusta en el Panel de control.
Pero lo que nos interesa en este caso es que el sistema genera nuevos mensajes WM_KEYDOWN o , sin los correspondientes mensajes de tecla liberada. Es más, cada uno de los mensajes puede corresponder a una pulsación, si el sistema es lo bastante rápido para procesar cada pulsación individual; o a varias, si se acumulan repeticiones entre dos mensajes consecutivos.
Para saber cuantas repeticiones de tecla están asociadas a un mensaje de tecla pulsada hay que examinar el cámpo de repetición del parámetro *lParam*.
El código de escaneo, como comentamos antes, es dependiente del dispositivo, y por lo tanto, generalmente no tiene utilidad para nosotros.
El bit de tecla extendida indica si se trata de una tecla específica de un teclado extendido. Generalmente, los ordenadores actuales siempre usan un teclado extendido.
El bit de contexto siempre es cero en los mensajes WM_KEYDOWN y WM_KEYUP, en el caso de los mensajes WM_SYSKEYDOWN y WM_SYSKEYUP será 1 si la tecla Alt está pulsada.
El bit de estado previo indica si la tecla estaba pulsada antes de enviar el mensaje, 1 si lo estaba, 0 si no lo estaba.
Y el bit de transición siempre es 0 en el caso de mensajes WM_KEYDOWN y WM_SYSKEYDOWN, y 1 en el caso de WM_KEYUP y WM_SYSKEYUP.
Cuando nuestra aplicación necesite procesar los mensajes de pulsación de tecla de sistema, debemos tener cuidado de pasarlos a la función DefWindowProc para que se procesen por el sistema. No lo hacemos esto, nuestra aplicación no responderá al menú desde el teclado, mediante combinaciones Alt+tecla.
Los mensajes de pulsación de tecla se usarán cuando queramos tener un control bastante directo del teclado, generalmente no nos interesa tanto control, y los mensajes de carácter serán suficientes.

```
        case WM_PAINT:
           hdc = BeginPaint(hwnd, &ps);
           SetBkColor(hdc,
GetSysColor(COLOR_BACKGROUND));
           for(i = 0; i < nLineas; i++)
              TextOut(hdc,   10,   i*20,   lista[i],
strlen(lista[i]));
           EndPaint(hwnd, &ps);
           break;
        case WM_KEYDOWN:
           for(i = nLineas; i > 0; i--)
              strcpy(lista[i], lista[i-1]);
           if(nLineas < 39) nLineas++;
           kd.lParam = lParam;
```

```
            sprintf(lista[0],   "Tecla   %d   pulsada
Rep=%d "
               "[Ext:%d Ctx:%d Prv:%d Trn:%d]",
               (int)wParam,            kd.repeticion,
kd.extendida,
               kd.contexto,               kd.previo,
kd.transicion);
            InvalidateRect(hwnd, NULL, TRUE);
            break;
         case WM_KEYUP:
            for(i = nLineas; i > 0; i--)
               strcpy(lista[i], lista[i-1]);
            if(nLineas < 39) nLineas++;
            kd.lParam = lParam;
            sprintf(lista[0], "Tecla %d liberada "
               "[Ext:%d Ctx:%d Prv:%d Trn:%d]",
               (int)wParam, kd.extendida,
               kd.contexto,               kd.previo,
kd.transicion);
            InvalidateRect(hwnd, NULL, TRUE);
            break;
```

Nombres de teclas

Una función que puede resultar útil en algunas circunstancias es GetKeyNameText, que nos devuelve el nombre de una tecla. Como parámetros sólo necesita el parámetro *lParam* entregado por un mensaje de pulsación de tecla, un búffer para almacenar el nombre y el tamaño del búffer:

```
char texto[128], cad[64];
...
         case WM_KEYDOWN:
            GetKeyNameText(lParam, cad, 64);
            sprintf(texto, "Tecla %s (%d) pulsada",
               cad, (int)wParam);
            break;
```

El bucle de mensajes

Es el momento de comentar algo sobre el bucle de mensajes que estamos usando desde el principio de este texto:

```
    while(TRUE == GetMessage(&mensaje, NULL, 0, 0))
    {
        /* Traducir mensajes de teclas virtuales a
mensajes de caracteres */
        TranslateMessage(&mensaje);
        /*  Enviar  mensaje  al  procedimiento  de
ventana */
        DispatchMessage(&mensaje);
    }
```

Me refiero a la función TranslateMessage, que como dice el comentario, traduce los mensajes de pulsaciones de teclas a mensajes de carácter. Si nuestra aplicación debe procesar los mensajes de pulsaciones de teclas no debería llamar a esta función en el bucle de mensajes. De todos modos, los mensajes de pulsación de teclas parecen llegar en los dos casos, pero no es mala idea seguir la recomendación del API en este caso.

Mensajes de carácter

Si usamos la función TranslateMessage, cada mensaje WM_KEYDOWN se traduce en un mensaje WM_CHAR o WM_DEADCHAR; y cada mensaje WM_SYSKEYDOWN a un mensaje WM_SYSCHAR o WM_SYSDEADCHAR.
Generalmente ignoraremos todos estos mensajes, salvo WM_CHAR. Los mensajes WM_SYSCHAR y WM_SYSDEADCHAR se usan por Windows para acceder de forma rápida a menús, y no necesitamos procesarlos. En cuanto al mensaje WM_DEADCHAR, notifica sobre caracteres de teclas muertas, y generalmente, tampoco resultará interesante procesarlos.

Teclas muertas

Veamos qué es este curioso concepto de tecla muerta. Las teclas muertas son aquellas que no generan un carácter por sí mismas, y necesitan combinarse con otras para formarlos. Por ejemplo, la tecla del acento (´), cuando se pulsa, no crea un carácter, es necesario pulsar otra tecla después para que eso ocurra. Si la tecla que se pulsa en segundo lugar genera un carácter que se puede combinar con la tecla muerta, se generará un único carácter, por ejemplo 'á'. Si no es así, se generan dos caracteres, el primero combinando la tecla muerta con un espacio, y el segundo con el carácter, por ejemplo "´b".
Cuando se pulse una tecla muerta, el mensaje que se genera por TanslateMessage puede ser WM_DEADCHAR o WM_SYSDEADCHAR, pero en cualquier caso, estos mensajes se puede ignorar, ya que el sistema los almacena internamente para generar los caracteres imprimibles.

```
        case WM_CHAR:
           switch((TCHAR) wParam) {
              case 13:
```

```
                // Procesar retorno de línea
                break;
            case 0x08:
                //  Procesar  carácter  de  retroceso
(borrar)
                break;
            default:
                // Cualquier otro carácter
                break;
        }
        InvalidateRect(hwnd, NULL, TRUE);
        break;
```

Estado de teclas

A veces nos interesa conocer el estado de alguna tecla concreto en el momento en que estamos procesando un mensaje procedente de otra pulsación de tecla. Por ejemplo, para tratar combinaciones de teclas como ALT+Fin o ALT+Inicio. Tenemos dos funciones para hacer esto.

Por una parte, la función GetAsyncKeyState nos dice el estado de una tecla virtual en el mismo momento en que la llamamos. Y la función GetKeyState nos da la misma información, pero en el momento en que se generó el mensaje que estamos tratando.

```
        case  WM_KEYDOWN:  //  CONTROL+Inicio  =  Borra
todo
            if(VK_HOME == (int)wParam) { // Tecla de
inicio
                if(GetKeyState(VK_CONTROL)  &&  0x1000)
{
                    nLinea=0;
                    nColumna=0;
                    lista[0][0] = 0;
                    InvalidateRect(hwnd, NULL, TRUE);
                }
            }
            break;
```

En este ejemplo, usamos la combinación CTRL+Inicio para borrar el texto que estamos escribiendo. Procesamos el mensaje WM_KEYDOWN, para detectar la tecla de [Inicio], y si cuando eso sucede, verificamos si también está pulsada la

tecla de [CTRL], para ello usamos la función GetKeyState y comprobamos si el valor de retorno tiene el bit de mayor peso activo, comparando con 0x1000.

Hot keys

He preferido no traducir el término "hot key", ya que me parece que es mucho más familiar que la traducción literal "tecla caliente". Una *hot key* es una tecla, o combinación de teclas, que tiene asignada una función especial y directa.
En Windows hay muchas hot keys predefinidas, por ejemplo, Ctrl+Alt+Supr sirve para activar el administrador de tareas, o la tecla de Windows izquierda en combinación con la tecla 'E', para abrir el explorador de archivos. Dentro de cada ventana o aplicación exiten más, por ejemplo, Alt+F4 cierra la ventana, etc.
Hay dos tipos de hot keys, uno es el de las asociadas a ventanas. Es posible asociar una tecla o combinación de teclas a una ventana, de modo que al pulsarla se activa esa ventana, estemos donde estemos, estas son las hot keys globales.
El otro tipo, que es el que vamos a ver ahora, son las hot keys de proceso, lo locales. Nuestra aplicación puede crear tantas de ellas como creamos necesario, procesarlas y, si es necesario, destruirlas.
Crear, o mejor dicho, registrar una hot key es sencillo, basta con usar la función RegisterHotKey. Esta función necesita cuatro parámetros. El primero es la ventana a la que estará asociada la hot key. El segundo parámetro es el identificador. El tercero son los modificadores de tecla, indica si deben estar presionadas las teclas de Control, Alt, Mayúsculas o Windows. Y el cuarto es el código de tecla virtual asociado a la hot key. Recordemos que los códigos de teclas virtuales de teclas correspondientes a caracteres son los propios caracteres, en el caso de letras, las mayúsculas.

```
        case WM_CREATE:
           hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
           color = GetSysColor(COLOR_BACKGROUND);
           RegisterHotKey(hwnd, ID_VERDE, 0, 'V');
           RegisterHotKey(hwnd,   ID_ROJO,   MOD_ALT,
'R');
           RegisterHotKey(hwnd,              ID_AZUL,
MOD_CONTROL, 'A');
           break;
```

Cada vez que se pulse la tecla o combinación de teclas correspondiente a una hot key, el sistema la busca entre las registradas, y envía un mensaje WM_HOTKEY a la ventana que la registró. Aunque esa ventana no esté activa, el mensaje será enviado, siempre que sea la única ventana que ha registrar esa combinación de teclas. En el parámetro *wParam* se recibe el identificador de la hot key.

```
        case WM_HOTKEY:
           switch((int)wParam) {
```

```
            case ID_VERDE:
               color = RGB(0,255,0);
               break;
            case ID_ROJO:
               color = RGB(255,0,0);
               break;
            case ID_AZUL:
               color = RGB(0,0,255);
               break;
         }
         InvalidateRect(hwnd, NULL, FALSE);
         break;
```

Finalmente, se puede desregistrar una hot key usando la función UnregisterHotKey, indicando la ventana para la que se registró, y el identificador.

```
      case WM_DESTROY:
         UnregisterHotKey(hwnd, ID_VERDE);
         UnregisterHotKey(hwnd, ID_ROJO);
         UnregisterHotKey(hwnd, ID_AZUL);
         PostQuitMessage(0);        /*  envía  un
mensaje WM_QUIT a la cola de mensajes */
         break;
```

Códigos de teclas virtuales

Los códigos virtuales de las teclas que generan caracteres son los códigos ASCII de esos caracteres, por ejemplo, el código virtual de la tecla [A] es el 'A', o en número, el 65. Para el resto de las teclas existen constantes definidas en el fichero "winuser.h". Las constantes definidas son:

Constante	Tecla	Constante	Tecla
VK_LBUTTON	Botón izquierdo de ratón	VK_RBUTTON	Botón derecho de ratón
VK_CANCEL		VK_MBUTTON	Botón central de ratón
VK_BACK		VK_TAB	Tabulador
VK_CLEAR		VK_RETURN	Retorno
VK_KANA		VK_SHIFT	Mayúsculas
VK_CONTROL	Control	VK_MENU	

VK_PAUSE	Pausa	VK_CAPITAL	Bloqueo mayúsculas
VK_ESCAPE	Escape	VK_SPACE	Espacio
VK_PRIOR	Página anterior	VK_NEXT	Página siguiente
VK_END	Fin	VK_HOME	Inicio
VK_LEFT	Flecha izquierda	VK_UP	Flecha arriba
VK_RIGHT	Flecha derecha	VK_DOWN	Flecha abajo
VK_SELECT		VK_PRINT	Imprimir pantalla
VK_EXECUTE		VK_SNAPSHOT	
VK_INSERT	Insertar	VK_DELETE	Suprimir
VK_HELP	Ayuda	VK_LWIN	Windows izquierda
VK_RWIN	Windows derecha	VK_APPS	Menú de aplicación
VK_NUMPAD0	'0' numérico	VK_NUMPAD1	'1' numérico
VK_NUMPAD2	'2' numérico	VK_NUMPAD3	'3' numérico
VK_NUMPAD4	'4' numérico	VK_NUMPAD5	'5' numérico
VK_NUMPAD6	'6' numérico	VK_NUMPAD7	'7' numérico
VK_NUMPAD8	'8' numérico	VK_NUMPAD9	'9' numérico
VK_MULTIPLY	Multiplicar	VK_ADD	Sumar
VK_SEPARATOR		VK_SUBTRACT	Restar
VK_DECIMAL	Punto decimal	VK_DIVIDE	Dividir
VK_F1	F1	VK_F2	F2
VK_F3	F3	VK_F4	F4
VK_F5	F5	VK_F6	F6
VK_F7	F7	VK_F8	F8
VK_F9	F9	VK_F10	F10
VK_F11	F11	VK_F12	F12

VK_F13	F13	VK_F14	F14
VK_F15	F15	VK_F16	F16
VK_F17	F17	VK_F18	F18
VK_F19	F19	VK_F20	F20
VK_F21	F21	VK_F22	F22
VK_F23	F23	VK_F24	F24
VK_NUMLOCK	Bloqueo numérico	VK_SCROLL	Bloqueo desplazamiento
VK_LSHIFT	Mayúsculas izquierdo	VK_RSHIFT	Mayúsculas derecho
VK_LCONTROL	Control izquierdo	VK_RCONTROL	Control derecho
VK_LMENU		VK_RMENU	
VK_PROCESSKEY		VK_ATTN	
VK_CRSEL		VK_EXSEL	
VK_EREOF		VK_PLAY	
VK_ZOOM		VK_NONAME	
VK_PA1		VK_OEM_CLEAR	

Las teclas sin descripción no están en mi teclado, de modo que no he podido averiguar a qué corresponden.

Capítulo 35 Cadenas de caracteres

Windows trata las cadenas de caracteres de un modo algo distinto a como lo hacen las funciones estándar C. Esto se debe a que Windows maneja varios conjuntos de caracteres: ANSI, que son los que ya conocemos, como caracteres de ocho bits y Unicode, que son de dos bytes.

También puede manejar, diferentes formas de comparar y ordenar cadenas, diferentes configuraciones de idioma, que afectan a la forma de representar mayúsculas y minúsculas, o de comparar caracteres, etc. Por ejemplo, en español, la letra 'ñ' es mayor que la 'n' y menor que la 'o'. En inglés ni siquiera existe esa letra. Otro ejemplo, si intentamos obtener la mayúscula de la letra 'ñ' usando funciones estándar, el resultado no será la 'Ñ'.

Recursos de cadenas

Al igual que podemos crear recursos para mapas de bits, menús, iconos, etc, también podemos crearlos para almacenar cadenas y leerlas desde la aplicación. Esto tiene varias ventajas y aplicaciones.

Los recursos de una aplicación pueden ser modificados por un editor adecuado sin modificar la parte ejecutable de una aplicación. Esto permite traducir una

aplicación a distintos idiomas sin tener que compilar la aplicación ni tener que compartir el código fuente.
Es más, podemos crear nuestras aplicaciones para que sean multilenguaje, de modo que usen las cadenas adecuadas según la configuración de la aplicación.

Fichero de recursos

Lo primero que debemos crear es una tabla de cadenas (stringtable) dentro del fichero de recursos, esto se hace mediante la sentencia STRINGTABLE:

```
STRINGTABLE
BEGIN
    ID_TITULO,    "Título de la aplicación"
    ID_SALUDO,        "Hola, estoy preparado para
empezar."
    ID_DESPEDIDA,    "Gracias    por    usar    esta
aplicación."
END
```

Como se ve, una tabla de cadenas empieza con la sentencia STRINGTABLE y a continuación, entre un bloque *BEGIN-END* una lista de identificadores y cadenas entre comillas, separadas con una coma. En este caso, como suele ser nuestra costumbre, los identificadores son etiquetas definidas en nuestro fichero de cabecera de identificadores, aunque podría tratarse de números enteros de 16 bits.
El objetivo es hacer nuestra aplicación tan independiente del idioma como sea posible. Esto significa que en la aplicación no deberían aparecer cadenas literales, sino que deben usar cadenas de recurso. De este modo, sólo con traducir las cadenas del fichero de recursos, todos los literales usados en la aplicación cambiarán de idioma. Esto evita tener que repasar todos los ficheros fuente buscando literales para sustituirlos, y tener que compilar la aplicación de nuevo.

Cargar cadenas desde recursos

Para obtener una cadena desde un recurso se usa la función LoadString. Esta función necesita cuatro parámetros. El primero es un manipulador de instancia, generalmente a la misma instancia de nuestra aplicación, aunque como veremos en capítulos más avanzados, también podemos obtener cadenas desde otros módulos, o desde DLLs. El segundo parámetro es el identificador de la cadena, el tercero el búfer donde se recibe la cadena leída, y el cuarto el tamaño de dicho búfer.
Por ejemplo:

```
    static HINSTANCE hInstance;
    char mensaje[64];
    char titulo[64];
...
        case WM_CREATE:
```

```
            hInstance   =   ((LPCREATESTRUCT)lParam)-
>hInstance;
            LoadString(hInstance,  ID_TITULO,  titulo,
64);
            LoadString(hInstance, ID_SALUDO, mensaje,
64);
            MessageBox(hwnd, mensaje, titulo, MB_OK);
            break;
```

Funciones para cadenas

Algunas funciones estándar C tienen una versión repetida en el API de Windows. En el caso de las siguientes funciones, están mejoradas para manipular cadenas Unicode:

Windows	C estándar
lstrcat	strcat
lstrcmp	strcmp
lstrcmpi	strcmpi
lstrcpy	strcpy
lstrlen	strlen

Por ejemplo, la versión Windows de **strlen**, lstrlen siempre calcula la longitud de una cadena en caracteres, independientemente de si los caracteres son de uno o dos bytes.

```
            // Comparar cadenas:
            if(lstrcmp("Niño", "Ñape") < 0)
               lstrcpy(mensaje,  "Niño  es  menor  que
Ñape");
            else
               lstrcpy(mensaje,  "Niño  es  mayor  que
Ñape");
            TextOut(hdc,     10,     280,     mensaje,
strlen(mensaje));
            if(lstrcmp("Ola", "Ñape") < 0)
               lstrcpy(mensaje,  "Ola  es  menor  que
Ñape");
            else
               lstrcpy(mensaje,  "Ola  es  mayor  que
Ñape");
```

```
            TextOut(hdc,     10,     300,     mensaje,
strlen(mensaje));
```

La salida de este fragmento de programa es la que cabría esperar:

```
Niño es menor que Ñape
Ola es mayor que Ñape
```

Es decir, la función considera que la letra 'Ñ' está entre la 'N' y la 'O', como realmente ocurre en español. Siempre y cuando nuestro Windows esté instalado en español, o el usuario haya seleccionado ese idioma, claro.
Otros casos donde es necesario crear nuevas funciones es cuando necesitamos operar con caracteres dentro de cadenas. Como en Windows los caracteres pueden ser de uno o dos bytes, moverse a lo largo de una cadena puede no ser siempre tan directo como usando cadenas C estándar. Disponemos de dos funciones para movernos dentro de cadenas: CharNext para avanzar al siguiente carácter de una cadena, y CharPrev para retroceder al anterior.
Algo parecido pasa con la conversión de mayúsculas a minúsculas, y viceversa. En este caso disponemos de cuatro funciones:

Función	Descripción
CharLower	Convertir un carácter o una cadena a minúsculas.
CharLowerBuff	Convertir un carácter o una cadena a minúsculas.
CharUpper	Convertir un carácter o una cadena a mayúsculas.
CharUpperBuff	Convertir un carácter o una cadena a mayúsculas.

```
    static          char          alfabeto[]          =
"abcdefghijklmnñopqrstuvwxyz áéíóúëü ç";
...
            // Mayúsculas y minúsculas
            CharLower(alfabeto);
            TextOut(hdc,     10,     220,     alfabeto,
strlen(alfabeto));
            CharUpper(alfabeto);
            TextOut(hdc,     10,     240,     alfabeto,
strlen(alfabeto));
```

También en este caso el resultado es el esperado, las letras se convierten a mayúscula y minúscula correctamente, aunque se trate de caracteres con acentos, diéresis o tildes:

```
abcdefghijklmnñopqrstuvwxyz áéíóúëü ç
```

```
ABCDEFGHIJKLMNÑOPQRSTUVWXYZ ÁÉÍÓÚËÜ Ç
```

Otro grupo de funciones estándar que se ven afectadas por el modo de trabajar en Windows son las del grupo de "ctype". En este caso tenemos otras cuatro funciones:

Función	Descripción
IsCharAlpha	Verificar si un carácter es alfabético.
IsCharAlphaNumeric	Verificar si un carácter es alfanumérico.
IsCharLower	Verifica si un carácter está en minúscula.
IsCharUpper	Verifica si un carácter está en mayúscula.

Por último, tenemos un par de funciones que sustituyen a las funciones estándar **sprintf** y **vsprintf**. Se trata de wsprintf y wvsprintf.

```
            for(i = 0; i < 10; i++) {
               wsprintf(mensaje,  "Cadena  formateada
%d: valor %c",
                  i+1, alfabeto[i]);
               TextOut(hdc,   10,   i*20,   mensaje,
strlen(mensaje));
            }
```

En Windows debemos usar las variantes del API, ya que están preparadas para trabajar con cadenas y caracteres Unicode, algo que las funciones estándar no pueden hacer.

Hay que tener en cuenta que estas funciones no aceptan las mismas cadenas de formato que **sprintf** o **vsprintf**, por ejemplo, no sirven para valores en punto flotante o punteros. Sin embargo, tienen más opciones para cadenas Unicode/ANSI.

Capítulo 36 Aceleradores

Los aceleradores son atajos para los menús. Lo normal y deseable es que nuestras aplicaciones proporcionen aceleradores para las opciones más frecuentes de los menús.

Un acelerador es una pulsación de tecla, o de teclas, que producen el mismo efecto que una selección en un menú. Windows detecta la pulsación y convierte el menaje de teclado a un mensaje WM_COMMAND o WM_SYSCOMMAND.

Cuando el usuario se familiariza con los aceleradores de teclado puede ahorrar mucho tiempo al activar comandos, ya que es mucho más rápido pulsar una tecla que activar una opción de menú, ya sea mediante el teclado o el ratón.

Recursos de aceleradores

Como ya hemos visto con los menús, cuadros de diálogo, cadenas, etc, en el caso de los aceleradores también podemos crearlos como un recurso, y cargarlos por la aplicación cuando se necesiten.

Fichero de recursos

Para crear una tabla de aceleradores se usa la sentencia ACCELERATORS:

```
aceleradores ACCELERATORS
BEGIN
  VK_F1, CM_OPCION1, VIRTKEY  /* F1 */
  "^C",  CM_SALIR             /* Control C */
  "K",   CM_OPCION2           /* K */
  "k",   CM_OPCION3, ALT      /* Alt k */
  0x34,  CM_OPCION4, ASCII    /* 4 */
  VK_F2, CM_OPCION5, ALT, SHIFT, VIRTKEY     /* Alt
Mays F2 */
  "1",    CM_OPCION6, ALT, CONTROL, VIRTKEY   /* Alt
Control 1 */
END
```

Si queremos que nuestros aceleradores no distingan mayúsculas de minúsculas, es mucho mejor definirlos a partir de teclas virtuales.

Cargar aceleradores desde recursos

Para cargar los aceleradores desde un recurso se usa la función LoadAccelerators. Como ya viene siendo corriente con este grupo de funciones, esta también necesita dos parámetros. El primero es un manipulador de la instancia del módulo que contiene el recurso, el segundo, es el identificador de recurso.

```
HACCEL hAcelerador = LoadAccelerators(hThisInstance,
"aceleradores");
```

Bucle de mensajes para usar aceleradores

Para que nuestra aplicación reciba mensajes cuando se pulsan las teclas que definen los aceleradores hay que modificar el bucle de mensajes. Los mensajes WM_KEYDOWN y WM_SYSKEYDOWN deben traducirse a mensajes WM_COMMAND y WM_SYSCOMMAND, y para ello hay que usar la función TranslateAccelerator:

```
    while(TRUE == GetMessage(&mensaje, NULL, 0, 0))
    {
        /* Traducir mensajes de teclas a mensajes de
acelerador */
```

```
        if(!TranslateAccelerator(hwnd,  hAcelerador,
&mensaje)) {
           /* Traducir mensajes de teclas virtuales
a mensajes de caracteres
              sólo  si  TranslateAccelerator  regresa
con nulo */
           TranslateMessage(&mensaje);
        }
        /*  Enviar  mensaje  al  procedimiento  de
ventana */
        DispatchMessage(&mensaje);
    }
```

Crear tablas de aceleradores sin usar recursos

Se usa la función CreateAcceleratorTable para crear tablas de aceleradores durante la ejecución, a partir de un array de estructuras ACCEL. El manipulador de aceleradores obtenido mediante esta función se puede usar igual que el obtenido por la función LoadAccelerators.
Un detalle importante: las tablas de aceleradores creadas mediante CreateAcceleratorTable se deben destruir antes de terminar la aplicación mediante una llamad a la función DestroyAcceleratorTable. Esto no es necesario cuando se usa la función LoadAccelerators.
En general usaremos aceleradores de recursos, ya que son más fáciles de manejar. Sin embargo, este procedimiento nos permitiría, por ejemplo, crear aceleradores definidos por el usuario.

Combinar aceleradores y menús

En general, los aceleradores estarán ligados a opciones de menú, pero no hay nada que informe al usuario sobre los aceleradores disponibles, salvo que nosotros lo indiquemos directamente.
Una forma fácil de hacerlo es añadir la información del acelerador al ítem del menú. Seguro que has notado que algunas opciones de menú tienen una combinación de teclas a su derecha, por ejemplo, en el menú de sistema de cualquier ventana la última opción es la de "Cerrar", y a su derecha aparece el texto "Alt+F4". Eso es un acelerador.
Nosotros podemos hacer lo mismo con nuestros menús. Es muy fácil añadir información a la derecha del texto de un ítem, basta con insertar la secuencia "\a", y a continuación el texto del acelerador. El texto después de la secuencia "\a" se justifica a la derecha:

```
menu MENU
BEGIN
 POPUP "&Principal"
 BEGIN
  MENUITEM "Opción &1\aF1", CM_OPCION1
  MENUITEM "Opción &2\aK", CM_OPCION2
  MENUITEM "Opción &3\aAlt+k", CM_OPCION3
  MENUITEM "Opción &4\a4", CM_OPCION4
  MENUITEM "Opción &5\aAlt+Mays+F2", CM_OPCION5
  MENUITEM "Opción &6\aAlt+Ctrl+1", CM_OPCION6
  MENUITEM SEPARATOR
  MENUITEM "&Salir\a^C", CM_SALIR
 END
END
```

Aceleradores globales

Existen varios aceleradores definidos a nivel global de Windows, nuestras aplicaciones deben intentar evitar definir esos aceleradores, aunque en principio no es imposible hacerlo, sencillamente no es aconsejable. Los aceleradores son:

Acelerador	Descripción
ALT+ESC	Cambia a la siguiente aplicación.
ALT+F4	Cierra una aplicación o ventana.
ALT+HYPHEN	Abre el menú de sistema de una ventana de documento.
ALT+PRINT SCREEN	Copia una imagen de la ventana activa al portapapeles.
ALT+SPACEBAR	Abre el menú de sistema para una ventana de aplicación.
ALT+TAB	Cambia a la siguiente aplicación.
CTRL+ESC	Cambia a la lista de tareas de Windows (menú de Inicio).
CTRL+F4	Cierra el grupo activo o ventana de documento.
F1	Arranca la ayuda si la aplicación la tiene.
PRINT SCREEN	Copia una imagen de la pantalla al portapapeles.
SHIFT+ALT+TAB	Cambia a la aplicación anterior. El usuario debe presionar Alt+Mays mientras presiona TAB.

Diferencia entre acelerador y menú

Usar un acelerador es prácticamente lo mismo que activar un ítem de un menú. En ambos casos se envía un mensaje WM_COMMAND o WM_SYSCOMMAND, y nuestro procedimiento de ventana lo procesará del mismo modo. En un caso el identificador será el que hemos usado para el acelerador, y en el otro el que hemos usado para el ítem del menú.
Sin embargo es posible que a veces nos interese saber si un comando procede de un acelerador o de un menú. Para saberlo podemos comprobar la presencia de un bit, el código de notificación del mensaje WM_COMMAND, que se proporciona en la palabra de mayor peso del parámetro *wParam*.

Capítulo 37 Menús 2

En el capítulo 5 tratamos el tema de los menús, pero de una manera superficial. La intención era dar unas nociones básicas para poder usar menús en nuestros primeros ejemplos. Ahora los veremos con más detalle, y estudiaremos muchas características que hasta ahora habíamos pasado por alto.
Los menús pueden tener, por ejemplo, mapas de bits a la izquierda del texto. También pueden comportarse como checkboxes o radiobuttons. Se pueden inhibir ítems. Podemos crear menús flotantes contextuales al pulsar con el ratón sobre determinadas zonas de la aplicación, etc.

Marcas en menús

Seguro que estás familiarizado con las marcas de chequeo que aparecen en algunos ítems de menú en casi todas las aplicaciones Windows. Es frecuente que se puedan activar o desactivar opciones, y que se pueda ver el valor actual de cada opción consultando menú. El funcionamiento es exactamente el mismo que el de los checkboxes y radiobuttons.
Hemos visto que los ítems de menú se comportan exactamente igual que los botones, pero hasta ahora sólo hemos usado los menús como un conjunto de "Pushbuttons", veremos qué otras opciones tenemos.

Menús como checkboxes

Recordemos que los checkboxes son uno o un conjunto de botones, cada uno de los cuales puede tener dos estados. Cada uno de los botones dentro de un conjunto de chekboxes puede estar marcado o no. De hecho, cada checkbox se comporta de un modo independiente, y sólo se agrupan conceptualmente, es decir, los grupos no son controlados por Windows.
Con menús podemos crear este efecto para cada ítem, añadiendo un mapa de bits a la izquierda que indique si la opción está marcada o no. Generalmente, cuando no lo esté, se eliminará la marca.

Disponemos de dos funciones para marcar ítems de menú. La más sencilla de usar es CheckMenuItem. Esta función necesita tres parámetros: el primero es el manipulador del menú, el segundo el identificador o la posición que ocupa el ítem, y el tercero dos banderas que indican si el segundo parámetro es un identificador o una posición y si el ítem se va a marcar o no.

Entonces, lo primero que necesitamos, es una forma de obtener un manipulador del menú de la ventana. Esto es sencillo: usaremos la función GetMenu, que nos devuelve el manipulador del menú asociado a una ventana.

Lo segundo será decidir si accedemos al ítem mediante un identificador o mendiante su posición. La primera opción es la mejor, ya que usando el identificador no necesitamos un manipulador al submenú concreto que tiene nuestro ítem. Eso siempre que los identificadores no estén duplicados, en ese caso necesitamos usar la segunda opción.

Otra cosa que necesitamos es averiguar si un ítem está o no marcado. Para ello podemos usar la función GetMenuState, que usa prácticamente los mismos parámetros que CheckMenuItem, salvo que el tercero sólo indica si el segundo es un identificador o una posición.

Por ejemplo, este sencillo código averigua si un ítem está o no marcado, y cambia el estado de la marca:

```
   if(GetMenuState(GetMenu(hwnd),          CM_OPCION1,
MF_BYCOMMAND) & MF_CHECKED)
      CheckMenuItem(GetMenu(hwnd),         CM_OPCION1,
MF_BYCOMMAND | MF_UNCHECKED);
   else
      CheckMenuItem(GetMenu(hwnd),         CM_OPCION1,
MF_BYCOMMAND | MF_CHECKED);
```

Sin embargo, la documentación del API de Windows dice que la función CheckMenuItem es obsoleta, aunque se puede seguir usando, y recomienda usar en su lugar la función SetMenuItemInfo. Esta función permite modificar otros valores, como veremos a lo largo de este capítulo.

Al tener más opciones, esta función es más complicada de usar. Necesita cuatro parámetros: el manipulador de menú, el identificador o posición del ítem, un tercer parámetro que indica si el segundo es un identificador o una posición y un puntero a una estructura MENUITEMINFO.

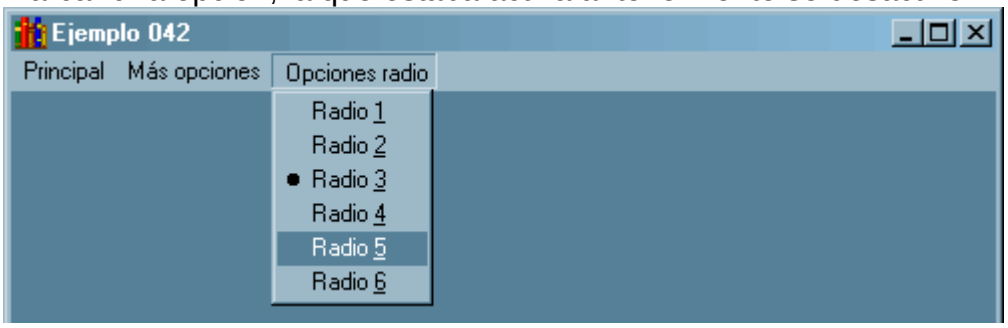
Esta estructura contiene campos que indican qué valores queremos modificar, y otros campos para indicar los nuevos valores. En nuestro caso, queremos modificar el valor de chequeo, por lo tanto, asignaremos el valor MIIM_STATE al campo *fMask* y al campo *fState* el valor apropiado: MFS_CHECKED o MFS_UNCHECKED.
La misma estructura se usa para recuperar valores de un ítem de menú, pero con la función GetMenuItemInfo, el valor del campo *fState* nos dirá si el ítem está o no marcado.

```
   MENUITEMINFO infoMenu;
...
   infoMenu.cbSize = sizeof(MENUITEMINFO);
   infoMenu.fMask = MIIM_STATE;
   GetMenuItemInfo(GetMenu(hwnd), CM_OPCIONA, FALSE,
&infoMenu);
   if(infoMenu.fState & MFS_CHECKED)
      infoMenu.fState = MFS_UNCHECKED;
   else
      infoMenu.fState = MFS_CHECKED;
   SetMenuItemInfo(GetMenu(hwnd), CM_OPCIONA, FALSE,
&infoMenu);
```

Por supuesto, a lo largo del programa siempre podremos consultar el estado de estos ítems, de modo que sabremos qué opción ha activado el usuario cuando lo necesitemos. Además, los items sigen generando mensajes WM_COMMAND, así que podemos procesarlos cuando sean modificados, en lugar se usarlos como simples editores de opciones.

Menús como radiobuttons

Cuando tenemos un grupo de opciones de las que sólo una de ellas puede estar activa, estamos ante un conjunto de radiobuttons. Estos botones sí deben estar agrupados, y se necesitan al menos dos de ellos en un grupo. En este caso, Windows sí puede gestionar el grupo automáticamente, para asegurarse que al marcar una opción, la que estaba activa anteriormente se desactive.



En este caso, procesar estos ítems es más simple, una única función bastará para gestionar todo un grupo de radioitems: CheckMenuRadioItem. Como primer parámetro tenemos el manipulador de menú, el segundo es el identificador o

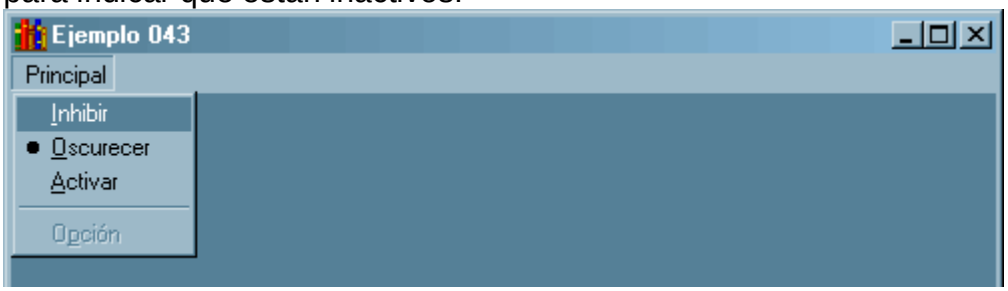
posición del primer ítem del grupo, el tercero el del último ítem del grupo, el cuarto el del ítem a marcar y el quinto indica si los parámetros segundo a cuarto son identificadores o posiciones.
Este ejemplo marca la opción 3 dentro de un grupo de 1 a 6. Automáticamente elimina la marca del ítem que la tuviese previamente:

```
   CheckMenuRadioItem(GetMenu(hwnd),CM_RADIO1,
CM_RADIO6,
      CM_RADIO3, MF_BYCOMMAND);
```

Al usar esta función, la marca normal (V) se sustituye por la del círculo negro.
Podemos seguir usando las funciones GetMenuState o mejor, GetMenuItemInfo, para averiguar qué ítem es el activo en un momento dado.

Inhibir y oscurecer ítems

También frecuente que en determinadas circunstancias queramos que algunas opciones no estén disponibles para el usuario, ya sea porque no tienen sentido, o por otra razón. Por ejemplo, esto es lo que pasa con la opción de "Maximizar" del menú de sistema cuando la ventana está maximizada.
En ese sentido, los ítems pueden tener tres estados distintos: activo, inhibido y oscurecido. Hasta ahora sólo hemos trabajado con ítems activos. Los inhibidos tienen el mismo aspecto para el usuario, pero no se pueden seleccionar. Los oscurecidos además de no poderse seleccionar, aparecen en gris o difuminados, para indicar que están inactivos.



Podemos cambiar el estado de acceso de un ítem usando la función EnableMenuItem, o mejor, con la función SetMenuItemInfo. Aunque la documentación del API dice que la primera está obsoleta, se puede seguir usando si no se necesitan otras características de la segunda.
La función EnableMenuItem necesita tres parámetros, el primero es el manipulador de menú, el segundo su identificador o posición, y el tercero indica si el segundo es un identificador o una posición y el estado que se va a asignar al ítem.
En este ejemplo se usa la función EnableMenuItem para inhibir y oscurecer un ítem, y la función SetMenuItemInfo para activarlo, de modo que se ilustran los dos modos de realizar esta tarea.

```
   MENUITEMINFO infoMenu;
...
```

```
   switch(LOWORD(wParam)) {
      case CM_INHIBIR:
         EnableMenuItem(GetMenu(hwnd),CM_OPCION,
MF_DISABLED | MF_BYCOMMAND);

CheckMenuRadioItem(GetMenu(hwnd),CM_INHIBIR,
CM_ACTIVAR, CM_INHIBIR, MF_BYCOMMAND);
         break;
      case CM_OSCURECER:
         EnableMenuItem(GetMenu(hwnd),CM_OPCION,
MF_GRAYED | MF_BYCOMMAND);

CheckMenuRadioItem(GetMenu(hwnd),CM_INHIBIR,
CM_ACTIVAR, CM_OSCURECER, MF_BYCOMMAND);
         break;
      case CM_ACTIVAR:
         infoMenu.cbSize = sizeof(MENUITEMINFO);
         infoMenu.fMask = MIIM_STATE;
         infoMenu.fState = MFS_ENABLED;
         SetMenuItemInfo(GetMenu(hwnd),    CM_OPCION,
FALSE, &infoMenu);

CheckMenuRadioItem(GetMenu(hwnd),CM_INHIBIR,
CM_ACTIVAR, CM_ACTIVAR, MF_BYCOMMAND);
         break;
```

Lo mismo se puede hacer con ModifyMenu, aunque esta función es obsoleta y se desaconseja su uso.