LINUX • JAVA • VISUAL BASIC • DEMOSCENE • VISUAL C++ • PROGRAMACIÓN 3D • SQL

PROGRAMACIÓN ACTUAL

AÑO 2 • NÚMERO 16

PORTUGAL 1250 ESC (CONT)

SÓLO 995 ptas

MIRRORS EN LINUX
COMO DOS GOTAS DE AGUA

SECCIÓN NIVEL BÁSICO
Programación del coprocesador en **ENSAMBLADOR**, En **PROGRAMACIÓN C++** herencia múltiple, repetida y restringida

SECCIÓN BAJO NIVEL
Control de cámaras en **ENTORNOS 3D**, Tipos de datos comunes en **PROGRAMACIÓN GRÁFICA** para **Windows**, En **DEMOSCENE** relleno de polígonos

SECCIÓN LAS EMPRESAS DEMANDAN
Tratamiento de errores en **VISUAL BASIC 5.0**, Todo sobre los controles ActiveX en **VISUAL C++**, Objetos DAO en **BASES DE DATOS**, Gestión de usuarios y recursos en **Windows NT**

CAMPUS ACTUAL
El arranque del kernel en **FORO LINUX**, Encaminamiento en redes TCP/IP en **RED LOCAL**, Programación en shell en **SISTEMAS OPERATIVOS**

ANÁLISIS ACTUAL
Netscape Mozilla y la historia de su liberación a fondo

EN EL CD ROM

PROGRAMACIÓN
- Amulet V3
- Dolphin Smalltalk 1.0
- GlowCode 1.0
- Java Plug-In 1.1
- Ozzino Studio 2.13
- Psyral Phobia 2.0
- Vodoo Coder 1.1
- W3D 1.0

UTILIDADES LINUX
- Binstats 1.01
- Kernel Linux 2.0.34
- Midnight Commander 4.1.35
- MySQL 3.21.30
- X11Amp 0.7
- SQLkit

CURSO DE PROGRAMACIÓN
DELPHI PROGRAMACIÓN EMPRESARIAL

Prens Técnic@
00016
8 413042 087779

Funciones, parámetros y mensajes

En la anterior entrega dejamos en el aire el significado de los parámetros de las funciones del programa de ejemplo, así como el contenido de algunas estructuras utilizadas en el código. A continuación se va a tratar de ver todas estas cosas más otro sencillo ejemplo que introduzca nuevos conceptos.
Por supuesto, hoy vamos a tratar la parte más tediosa de Windows, su cantidad de funciones, parámetros y tipos de datos, aunque aquí se tratará de simplificar su comprensión todo lo posible.

En el presente artículo se va a tratar de dar una referencia de los tipos de datos más comunes en Windows, así como de las funciones más utilizadas, y sus parámetros, de manera que podamos consultarlos sin aprender de memoria las funciones de básicas de la API de Windows.

TIPOS DE DATOS MAS FRECUENTES EN WINDOWS

Los tipos de datos más frecuentes son:

- **HWND:** Es un *handle* a una ventana, como a la ventana principal.
- **HDC:** Es un *handle* a un contexto de dispositivo (para trazar gráficos mediante él, usando el interfaz de Windows).
- **UINT**: Es un *typedef* de un entero de 32 bits sin signo (*unsigned int*), muy común para devolver datos.
- **WPARAM/LPARAM**: Enteros de 32 bits. Parámetros de *WndProc*().
- **BOOL**: Es un entero de 32 bits booleano (0/1 - TRUE/FALSE).
- **BYTE**: Dato de 8 bits (valor 0-256, equivalente a *unsigned char*).
- **DWORD**: Entero de 32 bits sin signo (equivalente a *unsigned int*).
- **LONG**: Es un entero de 32 bits (*long*).
- **LPSTR, LPCSTR**: Un puntero a una cadena de caracteres o a una constante de cadena (LPSRT = *Long Pointer to STRing*, LPCSTR = *Long Pointer to Constant STRing*).
- **HICON, HCURSOR, HBRUSH**: *Handles* a un icono, cursor o brocha.
- **HBITMAP**: Referente a un *bitmap* (gráfico).
- **RECT**: Se usa para determinar rectángulos dentro de un área gráfica. Es una estructura que contiene 4 campos, que son las coordenadas (x1, y1, x2, y2) de un rectángulo.
- **POINT**: Estructura que permite identificar puntos. Contiene dos campos que son las coordenadas (x,y) de un punto (en pixels desde la esquina de la ventana (0,0)).

LA FUNCION REGISTERCLASSEX()

Como vimos en el número anterior, esta función registra una ventana a través de un clase de ventana *WNDCLASSEX* previamente inicializada:

```
// definimos la estructura de la ventana
  WNDCLASSSEX wcx;

// determinamos los valores de los campos
  wcx.style = (etc....)

// Registramos la clase de ventana ya preparada:
  if( !RegisterClassEx( &wcx ) )
            return( FALSE );
```

La variable que se le pasa a *RegisterClassEx()* es una estructura de tipo de tipo *WNDCLASSEX*, la cuál tiene la siguiente forma en los archivos de cabecera de Windows:

```
typedef struct tagWNDCLASSEX
{
UINT cbSize ;          // tamaño de la estructura.
UINT style ;           // Indica el tipo de ventana.
WNDPROC lpfnWndProc ;  // Puntero a la función del procedimiento
                       // de ventana WinProc().
int cbClsExtra ;       // Entero con información adicional.
int cbWndExtra ;       // Entero con información adicional.
HINSTANCE hInstance ;  // Handle a la instancia actual.
HICON hIcon ;          // Handle al icono a usar.
HCURSOR hCursor ;      // Handle al cursor a usar.
HBRUSH hbrBackground ; // Handle a la brocha a usar
                       // para trazar el fondo de la ventana.
LPCSTR lpszMenuName ;  // Puntero al nombre del menu.
LPCSRT lpszClassName ; // Puntero al nombre de la clase.
HICON hIconSm ;        // Handle al icono pequeño.
} WNDCLASSEX;
```

LOADICON(), LOADCURSOR() Y GETSTOCKOBJECT()

A la hora de rellenar los diferentes campos de la estructura *WNDCLASSEX* se usan las funciones *LoadIcon()*, *LoadCursor()* y *GetStockObject()*, cada de ellas una con una finalidad diferente.

1. *LoadIcon():* Sirve para cargar un icono, devolviendo un *handle* tipo HICON del mismo. En el código de inicialización lo usamos para cargar un icono del *stock* de Windows y asignárselo a la ventana que se está creando. Sus parámetros de llamada son los siguientes:

```
HICON LoadIcon(
 HINSTANCE hInstance,   // Handle de la instancia
 LPCTSTR lpIconName     // Puntero a la cadena con un nombre
                        // o un identificador de recursos.
        );
```

El parámetro *lpIconName* constituye un identificador de recursos, de los cuales los más habituales son los que pueden verse en la tabla 1.

2. *LoadCursor():* Sirve para cargar un cursor para el ratón, devolviendo el *handle* identificador del mismo. De la misma manera, lo usamos en *WNDCLASSEX* para definir el aspecto del cursor cuando nuestra aplicación se esté ejecutando (aunque también podemos cambiarlo en cualquier momento):

```
HCURSOR LoadCursor(
  HINSTANCE hInstance,   // Handle de la instancia.
  LPCTSTR lpCursorName   // Puntero a la cadena con un nombre
```

```
        // o un identificador de recursos.
    );
```

Los valores más habituales de lpCursorName pueden observarse también en la tabla 1.

3.- GetStockObject(): Es utilizado para un extraer un objeto gráfico del *stock* predefinido del GDI de Windows, permitiéndonos utilizar recursos incluidos dentro del propio Windows, como las brochas (*brush*), paletas (*palette*), fuentes (*font*), plumas (*pen*), etcétera.

```
HBRUSH GetStockObject(
  int fnObject      // Entero con el valor del objeto.
        );
```

En nuestro caso la hemos utilizado para extraer la brocha blanca con la que rellenar el fondo de nuestra ventana (indicándolo en *wcx.hbrBackGround*).

LA FUNCION CREATEWINDOWEX()

La función CreateWindowEx(), como se comentó en el número anterior, crea la ventana tras registrarla. Su prototipo es el siguiente:

```
HWND CreateWindowEx(
 DWORD dwExStyle,       // Estilo extendido de la ventana.
 LPCTSTR lpClassName,   // Puntero al nombre de la clase
 LPCTSTR lpWindowName,  // Puntero al título para la ventana.
 DWORD dwStyle,         // Estilo de la ventana.
 int x, int y,          // posición de la ventana (x,y).
 int nWidth,            // Ancho de la ventana.
 int nHeight,           // Alto de la ventana.
 HWND hWndParent,       // Handle a la ventana padre
 HMENU  hMenu,          // Handle al menú (NULL=no menú)
 HANDLE hInstance,      // Handle a la instancia actual.
 LPVOID lpParam         // Puntero a los datos de creación
);
```

De esta declaración podemos deducir fácilmente el significado de (*x,y*) y (*nWidth, nHeight*) como la posición y tamaño de la ventana en pixels (referidos a la esquina (0,0) del escritorio (o pantalla)). El parámetro *LpClassname* es el nombre de la clase con el que registramos la ventana, y *hInstance* es un *handle* a la instancia actual de la aplicación (cuando ejecutamos varias veces un mismo programa, cada ventana es una nueva instancia del mismo). Muchos de esos parámetros se ponen a NULL, como por ejemplo al no disponer de ventana padre.

Por otra parte, *dwExStyle* y *dwStyle* nos van a permitir elegir el aspecto de nuestra ventana de entre algunos valores predefinidos que pueden verse en la tabla 2.

TABLA 1

Identificadores de iconos más habituales :	
IDI_APPLICATION	Icono por defecto de la aplicación.
IDI_WINLOGO	Icono con el logotipo de Windows 95.
IDI_EXCLAMATION	Icono con el dibujo del símbolo (!).
IDI_HAND	Icono con el dibujo de STOP.
IDI_ASTERISK	Icono de la i de información.
IDI_QUESTION	Icono con el dibujo del símbolo (?).
Identificadores de cursores más habituales :	
IDC_ARROW	Cursor con la flecha estándar.
IDC_APPSTARTING	Cursor con la flecha y el reloj de arena.
IDC_WAIT	Cursor con el reloj de arena.
IDC_CROSS	Cursor con líneas onduladas.
IDC_IBEAM	Cursor de edición de texto.
IDC_NO	Cursor del stop (circulo+aspa).
IDC_UPARROW	Cursor con la flecha vertical.
Identificadores de brochas más habituales :	
WHITE_BRUSH	Brocha blanca.
BLACK_BRUSH	Brocha negra.
DKGRAY_BRUSH	Brocha gris oscuro.
GRAY_BRUSH	Brocha gris.
LTGRAY_BRUSH	Brocha gris claro.
NULL_BRUSH	Brocha transparente.

TABLA 2

Estilos más usuales de ventanas (Style)	
WS_OVERLAPPEDWINDOW	Crea una ventana estándar de Windows 95.
WS_OVERLAPPED	Crea una ventana superpuesta.
WS_TILEDWINDOW	Crea una ventana estándar de Windows 95.
WS_BORDER	Crea una ventana con borde.
WS_CAPTION	Crea una ventana con borde y barra de título.
WS_HSCROLL	Crea una ventana con barra de desplazamiento horizontal.
WS_ICONIC	Crea una ventana minimizada.
WS_MAXIMIZE	Crea una ventana maximizada.
WS_MINIMIZE	Crea una ventana que se minimiza..
WS_POPUP	Crea una ventana tipo pop-up (sin menú, título ni borde).
WS_TILED	Crea una ventana superpuesta.
WS_VISIBLE	Crea una ventana inicialmente visible.
WS_VSCROLL	Crea una ventana con barra de desplazamiento vertical.
CW_USEDEFAULT	Usar valores por defecto.
Estilos extendidos más usuales de ventanas (exStyle)	
WS_EX_OVERLAPPEDWINDOW	Crea una ventana estándar de Windows 95.
WS_EX_TRANSPARENT	Crea una ventana transparente.
WS_EX_TOOLWINDOW	Crea una ventana de herramientas.
WS_EX_ABSPOSITIO	Crea una ventana de posición fija.
WS_EX_CLIENTEDGE	Borde hundido (efecto 3d).
WS_EX_WINDOWEDGE	Borde elevado (efecto 3d).
WS_EX_TOPMOST	Crea una ventana siempre visible (TOP).
WS_EX_CONTEXTHELP	Crea una ventana con botón de ayuda.

CUADROS DE DIALOGO

Una de las funciones de ventana más útiles de Windows es la posibilidad de mostrar y gestionar cuadros de diálogo del estilo Sí/No, Aceptar/Cancelar, etcétera, muy útiles para mostrar información, resultado de errores (para poder *debuggear* la aplicación), pedir confirmaciones al usuario, permitiendo un rápido y sencillo control de flujo del programa.

El mes pasado pudimos ver en el listado 1 un ejemplo de cuadro de diálogo de información, que esperaba la pulsación del botón *Aceptar* para continuar la ejecución. La creación de uno de estos cuadros de diálogo es muy sencilla

gracias a *MessageBox()* y *MessageBoxEx* (como siempre, el prefijo Ex significa Extendido), funciones de la API de Windows para la gestión de los mismos. Los cuadros de mensaje pueden ser «*modales*» y «*modales de sistema*». La diferencia entre ambos radica en que los primeros no permiten acceder a otra ventana del programa hasta que la respuesta del usuario se produzca, aunque permite el acceso a otros programas distintos del nuestro (éstos son los cuadros por defecto). Por otra parte, los segundos no permiten el acceso a ninguna otra aplicación (incluida la nuestra) hasta que son respondidos. Veamos el prototipo de la función *MessageBox()*:

```
int MessageBox(
  HWND  hwnd,        // handle ventana
  LPCTSTR  lpText,   // frase información
  LPCTSTR  lpCaption, // título
  UINT  uType,       // Estilo del cuadro
);
```

De los cuatro parámetros, el primero es el *handle* a la ventana padre (que puede ser NULL si no queremos que pertenezca a ninguna ventana en concreto), el segundo (*lpText*) un puntero al texto que queremos que aparezca en él (Windows partirá las líneas si es necesario), pudiendo además incluir secuencias de escape como **\n**, etcétera. El tercero constituye el título de la ventana y el último (y más importante) el estilo del diálogo, es decir, los botones y el tipo de diálogo deseado. Los diferentes tipos pueden apreciarse en la tabla 3. Podemos usar varios de estos indicadores simultáneamente empleando el operador OR (|). Como resultado, *MessageBox* nos devuelve el botón que haya sido pulsado (ya sea con ratón o teclado), o cero en caso de error. En la tabla 3 pueden verse también los valores que puede devolver la función.
Un ejemplo del uso de *MessageBox* se ha implementado en el listado 1.

LAS MACROS LOWORD E HIWORD

LOWORD e HIWORD son dos funciones de tipo macro predefinidas para Windows que nos permiten extraer el WORD bajo y alto, respectivamente, de cualquier variable de 32 bits que se le especifique. Esto resulta muy útil muchas veces para obtener parámetros pasados a la función de procedimiento de ventana. Veamos como ejemplo una porción de código que nos permitiría saber el tamaño de nuestra ventana cuando cambia su tamaño (mensaje WM_SIZE, recibimos en *lParam* el nuevo ancho y alto):

```
int Ancho, Alto;

// ... código ...
case WM_SIZE :
 Ancho = LOWORD( lParam ) ;
 Alto  = HIWORD( lParam ) ;
 return(0) ;
```

Cuando el mensaje recibido es WM_SIZE, en *lParam* se obtiene el nuevo tamaño del área cliente de la ventana, estando en la parte alta de *lParam* la anchura y en el WORD bajo, la altura.

LISTADO 1

```
#include <windows.h>

int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE
hPrevInstance,
                                    LPSTR lpCmdLine, int nCmdShow )
{
int opcion;

  opcion = MessageBox( NULL,
          "Texto de información.",
              "Título del MessageBox.",
        MB_OKCANCEL | MB_ICONASTERISK );

  if( opcion == IDOK )
    MessageBox(NULL, "Ha pulsado ACEPTAR",
                    "Salir", MB_OK );
  else
    MessageBox(NULL, "Ha pulsado CANCELAR",
                    "Salir", MB_OK );
  return(0); }
```

TABLA 3

Estilos de cuadros de mensajes:	
MB_OK	El DialogBox contiene el botón Aceptar.
MB_OKCANCEL	El DialogBox contiene Aceptar y Cancelar.
MB_RETRYCANCEL	"" 2 botones: Reintentar y Cancelar.
MB_YESNO	"" 2 botones: Sí y No.
MB_YESNOCANCEL	"" 3 botones: Sí, No, y Cancelar.
MB_ABORTRETRYIGNORE	"" 3 botones: Anular, Reintentar e Ignorar.
MB_DEFBUTTON1	El primer botón es el seleccionado por defecto. Esta es la opción por defecto.
MB_DEFBUTTON2	El 2º botón es el botón seleccionado por defecto.
MB_DEFBUTTON3	El 3er botón es el seleccionado por defecto.
MB_DEFBUTTON4	El 4to botón es el seleccionado por defecto.
MB_HELP	Añade un botón de Ayuda al cuadro de mensaje.
MB_ICONASTERISK	Añade un icono con una letra *i* de información (de color azul en un círculo blanco).
MB_ICONERROR	Añade Un icono con una señal de stop (circulo con aspa blanca) al dialogbox.
MB_ICONEXCLAMATION	Añade un icono con un signo de admiración (!) en un triángulo amarillo al cuadro de mensaje.
MB_ICONQUESTION	Añade un icono con un signo de interrogación (?).
MB_ICONSTOP	Igual que MB_ICONERROR.
MB_ICONWARNING	Igual que MB_ICONEXCLAMATION.
MB_APPLMODAL	Indica que el cuadro de mensaje es modal.
MB_SYSTEMMODAL	Indica que el cuadro de mensaje es modal de sistema.
Valores de retorno de MessageBox():	
IDOK	Se pulsó el botón «Aceptar».
IDCANCEL	Se pulsó el botón «Cancelar».
IDABORT	Se pulsó el botón «Anular».
IDIGNORE	Se pulsó el botón «Ignorar».
IDRETRY	Se pulsó el botón «Reintentar».
IDYES	Se pulsó el botón «Sí».
IDNO	Se pulsó el botón «No».

LA FUNCION TEXTOUT(): TEXTO EN NUESTRA APLICACION

Una de las funciones más importantes para hacer nuestras primeras pruebas y programas es la salida de texto en pantalla, muy útil para

LISTADO 2

```
//————————————————————————————————————————
// EJEMPLO2.CPP - Programa de ejemplo bajo Windows95.
// Santiago Romero Iglesias AKA NoP / Compiler,
//————————————————————————————————————————
#include <windows.h>

//— Declaración de funciones del programa —————
int WINAPI WinMain( HINSTANCE, HINSTANCE, LPSTR, int );
LRESULT CALLBACK WndProc( HWND, UINT, WPARAM, LPARAM );
void MenuProc( HWND, UINT, WPARAM, LPARAM );

//— Declaración de variables del programa —————
char WindowName[] = "Default Window";
char WindowTitle[] = "Windows95 application";

//=== Función principal WinMain() ====================
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
  HWND hwnd;
  MSG msg;
  WNDCLASSEX wcx;

  // Definimos la estructura de clase:
  wcx.cbSize = sizeof( WNDCLASSEX );
  wcx.style = CS_HREDRAW | CS_VREDRAW;
  wcx.lpfnWndProc = WndProc;
  wcx.cbClsExtra = 0;
  wcx.cbWndExtra = 0;
  wcx.hInstance = hInstance;

  // icono, cursor, fondo e icono pequeño:
  wcx.hIcon = LoadIcon(NULL, IDI_WINLOGO);
  wcx.hCursor = LoadCursor(NULL, IDC_ARROW);
  wcx.hbrBackground = (HBRUSH) GetStockObject( WHITE_BRUSH );
  wcx.hIconSm = LoadIcon(NULL, IDI_WINLOGO);
  wcx.lpszClassName = WindowName;
  wcx.lpszMenuName = NULL;

  // Registramos la clase de ventana ya preparada:
  if( !RegisterClassEx( &wcx ) )
     return( FALSE );

  // Creamos la ventana con CreateWindowEx():
  hwnd = CreateWindowEx(
    WS_EX_OVERLAPPEDWINDOW,
    WindowName, WindowTitle,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,
    400, 300, NULL, NULL,
    hInstance, NULL
                 );

  // Comprobamos la creación de la ventana:
  if( !hwnd )
    return( FALSE );
  // Hacemos visible la ventana y la actualizamos:
  ShowWindow( hwnd, nCmdShow );
  UpdateWindow( hwnd );

  // Bucle de mensajes, envía los mensajes hacia WndProc
  while( GetMessage( &msg, NULL, 0, 0 ) )
  {
     TranslateMessage( &msg );
     DispatchMessage( &msg );
  }

  // devolvemos el valor recibido por PostQuitMessage().
  return( msg.wParam );
}

//=== Función del procedimiento de ventana WndProc() ====
LRESULT CALLBACK WndProc( HWND hwnd, UINT message,

WPARAM wParam, LPARAM lParam )
{
  HDC hdc;
  PAINTSTRUCT ps;
  TEXTMETRIC tm;
  RECT rect;
  int loop;
  static long cxChar, cyChar;

  switch( message )
  {
    // mensaje producido en la creación de la ventana
    case WM_CREATE:
       hdc = GetDC( hwnd );
       GetTextMetrics( hdc, &tm );
       cxChar = tm.tmAveCharWidth;
       cyChar = tm.tmHeight + tm.tmExternalLeading;
       ReleaseDC( hwnd, hdc );
       break;

    case WM_PAINT:
       hdc = BeginPaint( hwnd, &ps );
       GetClientRect( hwnd, &rect );
       for( loop=0; loop<10; loop++ )
         TextOut( hdc, 0,
                  loop*cyChar,
                  "ESTO ES UNA PRUEBA", 18 );
       EndPaint( hwnd, &ps );
       break;

    // mensaje producido al aumentar el tamaño de la ventana
    case WM_SIZE:
    // aqui habriamos de coger de nuevo el tamaño.
       break;
    // mensaje producido al cerrar la ventana
    case WM_DESTROY:
       PostQuitMessage( 0 );
       break;

    // resto de mensajes, dar una respuesta estándar.
                default:
       return( DefWindowProc( hwnd, message, wParam, lParam ) );
    }
    return(0);
}
```

debuggear la aplicación y obtener mensajes en las ventanas de la misma. Para ello se dispone de la función *TextOut* (aparte de la *DrawText* vista en el ejemplo del mes anterior):

```
TextOut(
        hdc,      // handle al dc
        x, y,     // coordenadas donde escribir
        psString, // puntero al texto
        iLenght   // longitud de la cadena
    );
```

Los parámetros a pasarle a la función son: un *handle* de contexto de dispositivo (que podemos obtener mediante *BeginPaint()* y bien, como veremos el mes que viene, mediante GetDC), las coordenadas (en pixels) donde imprimir la cadena, la propia cadena, y la longitud en caracteres de la misma. Es necesario incluir la longitud en caracteres pues esta función no busca el carácter END OF STRING (0) al final de la misma.

Por supuesto, necesitamos saber la anchura y altura en pixels de la fuente para poder escribir las líneas una debajo de la otra (o caracteres en posiciones concretas de la ventana, haciendo una «posición actual» a modo de cursor donde

escribir, como en MS-DOS. Para ello necesitamos saber la altura y anchura de cada carácter para la fuente que esté seleccionada para nuestra aplicación (la *System Font*, por defecto). El mejor momento de hacer esto es durante la creación de la aplicación, cogiendo la anchura y altura de cada carácter mediante la función *GetTextMetrics()*, como puede ver en el listado 2, donde se utilizan unas variables de tipo *static* (no desaparece su valor al salir de la función) llamadas *cxChar* y *cyChar*, que contienen el ancho y alto medio de cada carácter para la *system font*.

TEXTOUT Y WSPRINTF

La forma más sencilla de utilizar TextOut() es conjuntamente con la función *wsprintf* (similar a sprintf() de stdio.h). Esta función introduce cadenas y variables (admitiendo especificadores de formato) dentro de un *buffer* de texto (los concatena), devolviendo su longitud. A título de ejemplo:

```
char cadena[80];
int longitud;
// ... código ...
longitud = wsprintf(cadena, "El resultado es %d", total );
TextOut( hdc, 100, 100, cadena, longitud );
```

De esta manera, en nuestra aplicación podremos imprimir tanto cadenas de texto puro como valores numéricos de cualquier tipo, aprovechando además que wsprintf() nos devuelve el tamaño de la cadena creada.

EL BUCLE DE MENSAJES

El bucle de mensajes es sin duda uno de los puntos vitales de nuestro programa, ya que en él se recogen y traducen todos los mensajes de Windows a los parámetros que finalmente recibe la función *callback* de procedimiento de ventana.

```
while (GetMessage(&msg, NULL, 0, 0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}
return (msg.wParam) ;
```

Éste es un bucle que se repetirá hasta que el usuario salga de la aplicación, momento en el que recibiremos un cero y el *while* será *FALSE*. En caso de recibir un valor distinto de cero (*TRUE*), el bucle continúa. Hay que decir que Windows tiene una cola de mensajes, existiendo una continua llegada y traducción de los mismos.

```
BOOL GetMessage(
 LPMSG msg,     // Puntero a una estructura que
            // contendrá el mensaje recibido.
 HWND hwnd,     // Handle a la ventana que lo recibe.
 UINT wmsgmin,  // Nº identificador del primer mensaje.
 UINT wmsgmax   // Nº identificador del último mensaje.
            );
```

El tipo de dato LPMSG es (vayamos acostumbrándonos a la terminología «húngara» de Windows) un puntero largo a MSG (LP = *Long Pointer*):

```
typedef struct tagMSG
{
  HWND hwnd;      // Handle a la ventana destinataria.
  UINTmessage;    // Identificador del mensaje.
  WPARAM wParam;  // Información adicional.
  LPARAM lParam;  // Información adicional.
  DWORD time;     // Hora de envío del mensaje.
  POINT pt;       // Coordenadas del ratón.
} MSG;
```

WPARAM es el mismo tipo que UINT, y LPARAM es un *typedef* para LONG. La estructura tipo POINT contiene las coordenadas del ratón en el momento de recibir el mensaje, y está definida como:

```
typedef struct tagPOINT
{
    LONG x;       // coordenada x.
    LONG y;       // coordenada y.
} POINT;
```

Como puede verse en la línea *GetMessage(&msg, NULL, 0, 0)*, el *handle* a la ventana puede especificarse como NULL, y *lparam/wparam* a cero con el fin de obtener todos los mensajes para nuestra aplicación, no sólo para una determinada ventana (si hubiésemos creado diferentes ventanas hijas). Continuando con el comentario del bucle de mensajes, mediante *TranslateMessage()* traducimos algunos de estos mensajes, y mediante *DispatchMessage()* los enviamos a la función de procedimiento de ventana para que ésta realice su proceso.

OTRA MANERA DE RECOGER LOS MENSAJES

Bajo Windows tenemos otra posibilidad en vez de esperar a que Windows nos envíe un mensaje: continuar trabajando (aunque no nos envíe mensajes) recogiendo nosotros mismos todos los mensajes que se envíen en Windows y contestando tan sólo a los que nos atañen. Para ello puede utilizarse la función *PeekMessage()*, lo cual nos permitirá realizar otras acciones (como actualizar nuestro programa) una vez por cada mensaje que sea enviado en Windows.

```
while( 1 )
{
 HacerAlgo();
 if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ))
 {
  if( msg.message == WM_QUIT )
   return( msg.wParam );
  TranslateMessage( &msg );
  DispatchMessage( &msg );
 }
}
```

Mediante el bucle anterior hacemos que nuestra función vaya en busca de los mensajes, al mismo tiempo que ejecutamos código propio de nuestra aplicación (llamado en el ejemplo anterior de manera ilustrativa *HacerAlgo()*), que podría, en el caso de un juego, redibujar la pantalla, mover los *sprites* a distintas posiciones. Nótese que con esto estamos robándole tiempo a otras aplicaciones.

Email: sromero@arrakis.es
mcubas@arrakis.e

La próxima entrega

En la próxima entrega se tratará la manera de organizar el programa de Windows para que ejecute otras acciones aparte de gestionar los mensajes, así como se introducirá brevemente el GDI (tan sólo en lo que atañerá a un futuro uso conjunto con DirectX), y un timer simple para temporizar nuestras aplicaciones.