Finestre in Win32 (For Newbies)		
Data	by Spider	
ottobre 2001	***UIC's Home Page***	**Published by Quequero**
	Qualche mio eventuale commento sul tutorial :)))	
....	**Home page:** http://bigspider.cjb.net **E-mail:** spider_xx87@hotmail.com **Server IRC: irc.azzurra.it** **Canali: #crack-it #asm** **Nickname: ^Spider^**	....
Difficoltà	(X)**NewBies** ( )**Intermedio** ( )**Avanzato** ( )**Master**	

Con questo tutorial intendo dare uno sguardo a ciò che riguarda la creazione e l'uso di finestre in Windows a 32 bits.

Finestre in Win32
(For Newbies)
Written by Spider

Tools usati

- Una buona API reference. Essenziale per programmare in Win32. Una la trovate su http://itassembly.cjb.net nella sezione "Documenti vari".

Essay

In Windows la gestione di finestre e pulsanti è affidata alla cosiddetta GUI (Graphical User Interface). Questo ci permette di risparmiare grandi quantità di codice, evitandoci di dover ogni volta riscrivere tutto il codice per il design e la programmazione delle varie finestre; inoltre la presenza della GUI fa sì che tutti i programmi seguano quello standard, semplificando la vita agli utenti che così non devono imparare tutto da capo per ogni programma.

Per finestre non intendiamo soltanto quelle con la barra di sopra, i menu, il pulsantini per ingrandire, ridurre a icona e chiudere... Con il nome 'finestre' intendiamo anche le i pulsanti, le barre degli strumenti, le scritte (statics), le caselle di testo, e quasi tutti ciò che noi possiamo trovare all'interno dei programmi e di Windows stesso.

Windows ci mette a disposizione una gran quantità di API per la gestione della GUI. Se vogliamo creare una semplice finestra, possiamo seguire diversi approcci. Tuttavia abbiamo SEMPRE bisogno di un elemento fondamentale: la procedura di gestione dei messaggi, chiamata WndProc o WindowProc (anche se nel vostro programma potete chiamarla come vi pare, tanto quel nome lo vedrete solo voi :P). La WndProc riceve i messaggi che il sistema operativo manda alla nostra finestra. Ad esempio, se vogliamo intercettare il momento in cui un utente clicca sulla finestra, dobbiamo *processare* il messaggio WM_CLICK. La WndProc riceve 4 argomenti. Nell'ordine: *hWnd, uMsg, wParam* e *lParam*. *hWnd* contiene l'handle della finestra di cui ci stiamo occupando; un handle è un numero che, in windows, serve per identificare qualunque oggetto (finestre, brushes, files, ecc.). Ogni finestra avrà il suo handle che permetterà di distinguerla dalle altre.

Normalmente le applicazioni win32 DEVONO runnare in un loop infinito che continuerà fino alla chiusura del programma stesso. Questo loop deve essere simile a questo:

```
.WHILE TRUE
   invoke GetMessage, ADDR msg,NULL,0,0
   .BREAK .IF (!eax)
   invoke TranslateMessage, ADDR msg
   invoke DispatchMessage, ADDR msg
.ENDW
```

La chiamata a GetMessage non restituisce il controllo al nostro programma finché non ci sono messaggi destinati alla nostra applicazione. La funzione ritorna FALSE se è stato ricevuto il messaggio WM_QUIT, in modo che il programma potrà uscire dal loop e chiudersi.
TranslateMessage traduce la pressione di un tasto in un messaggio che sarà letto dalla successiva chiamata a GetMessage.
DispatchMessage si occupa di inviare il messaggio alla corretta window procedure.

Per creare una *finestra di dialogo* possiamo seguire due approcci principali. Diamo un'occhiata ai vari metodi e vediamo quali sono i pregi e i difetti di ogni approccio.

Primo approccio: CreateWindowEx

Il primo metodo, il più semplice, consiste nell'utilizzare la funzione API CreateWindowExA, che ci permette di creare un qualunque tipo di finestra. La CreateWindowExA. Dall'API Reference:

```
HWND CreateWindowEx(

DWORD dwExStyle,      // extended window style
LPCTSTR lpClassName,  // pointer to registered class name
LPCTSTR lpWindowName, // pointer to window name
DWORD dwStyle,        // window style
int x,                // horizontal position of window
int y,                // vertical position of window
int nWidth,           // window width
int nHeight,          // window height
HWND hWndParent,      // handle to parent or owner window
HMENU hMenu,          // handle to menu, or child-window identifier
HINSTANCE hInstance,  // handle to application instance
LPVOID lpParam        // pointer to window-creation data
);
```

Come vedete sono ben 12 parametri, ma in realtà è molto semplice. Vediamoli in dettaglio:

dwExStyle - Extended styles, ovvero degli stili di finestra particolari tipo WS_EX_TOPMOST, WS_EX_ACCEPTFILES, ecc. Se non vogliamo stili particolari per questo parametro dovremo passare NULL.

lpClassName - Il nome della classe che vogliamo creare. Ad esempio, se vogliamo creare un pulsante, questo parametro dovrà essere "Button"

lpWindowName - Il nome della finestra. Ad esempio, se creiamo una finestra di dialogo corrisponderà al titolo, o in un bottone al testo contenuto all'interno, ecc.

dwStyle - Gli stili standard della finestra. Consultare una API Reference per una lista completa. Tutte le costanti degli stili iniziano per WS_

x - La posizione orizzontale della finestra.

y - La posizione verticale della finestra.

nWidth - La larghezza della finestra.

nHeight - L'altezza della finestra.

hWndParent - L'handle della finestra genitore, ovvero quella che conterrà la finestra che noi vogliamo creare. Se dobbiamo creare una finestra di dialogo il genitore sarà il desktop; in tal caso è sufficiente passare NULL e il sistema operativo capirà che vogliamo una finestra figlia del desktop. :-)

hMenu - L'handle dell'eventuale menu che vorremo inserire nella nostra finestra. Se non vogliamo mettere menu, deve essere NULL.

hInstance - L'hInstance della nostra applicazione (ma va!...)

lpParam - Per ora vi basti sapere che se passate NULL funzionerà tutto :P lpParam serve per usi un po' più avanzati.

Spero vi sia sorta una domanda... Per creare un bottone si passa "Button" come lpClassName, per un testo si passa "Static"... e per una finestra di dialogo??? Semplice... Prima ci registriamo una *classe* che chiameremo come più ci piace... Poi la creiamo con CreateWindowEx, passando in lpClassName il nome che avevamo scelto. Per registrare una classe si utilizza RegisterClassEx. Suppongo che abbiate ancora più confusione di prima. Vediamo la descrizione di RegisterClassEx e poi facciamo un esempio che successivamente commenteremo.

```
ATOM RegisterClassEx(

CONST WNDCLASSEX *lpwcx // address of structure with class data
);
```

Si passa un solo parametro... Voi direte: ma è così semplice? No: il parametro passato è l'indirizzo di una struttura WNDCLASSEX, che ora andremo a vedere:

```
typedef struct _WNDCLASSEX { // wc
   UINT cbSize;
   UINT style;
   WNDPROC lpfnWndProc;
   int cbClsExtra;
   int cbWndExtra;
   HANDLE hInstance;
   HICON hIcon;
   HCURSOR hCursor;
   HBRUSH hbrBackground;
   LPCTSTR lpszMenuName;
   LPCTSTR lpszClassName;
   HICON hIconSm;
} WNDCLASSEX;
```

Un'altra mazzata di 12 elementi :) analizziamoli:

cbSize - La larghezza in bytes della struttura. E' necessario settarla correttamente, altrimenti non funzionerà un bel niente. Per sapere la larghezza della struttura possiamo utilizzare l'operatore SIZEOF.

style - Specifica lo stile/gli stili della classe. Per una lista completa consultate l'API Reference. I vari stili vanno combinati con l'operatore OR.

lpfnWindowProc - Punta alla Window Procedure della classe.

cbClsExtra e **cbWndExtra** - Non li ho mai utilizzati, non ho ben chiaro neanch'io quale sia il loro uso... consultate una API Reference =)

hInstance - L'hInstance del nostro programma :)

hIcon - L'icona della nostra classe.

hCursor - Il cursore della classe.

hbrBackGround - Il brush per il background della class.

lpszMenuName - Punta ad una stringa che identifica il nome del menu all'interno delle risorse.

lpszClassName - Specifica il nome della nostra classe, ovvero quello che utilizzeremo per creare la finestra con CreateWindowEx.

hIconSm - Handle di una icona piccola associata alla classe.

Quando non vogliamo passare un parametro ma vogliamo utilizzare il valore standard, è sufficiente passare NULL.

Bene... Dopo la teoria passiamo alla pratica :) ora faremo un esempio in cui venga utilizzato questo primo approccio.

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

.data
ClassName db "FirstWindow",0
AppName db "Wow! La nostra prima finestra!",0


.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?

.code
start:
   invoke GetModuleHandle, NULL
   mov hInstance,eax
   invoke GetCommandLine
   mov CommandLine,eax
   invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
   invoke ExitProcess,eax

WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
      LOCAL wc:WNDCLASSEX
      LOCAL msg:MSG
```

```
    mov wc.cbSize,SIZEOF WNDCLASSEX
    mov wc.style, CS_HREDRAW or CS_VREDRAW
    mov wc.lpfnWndProc, OFFSET WndProc
    mov wc.cbClsExtra,NULL
    mov wc.cbWndExtra,NULL
    push hInstance
    pop wc.hInstance
    mov wc.hbrBackground,COLOR_WINDOW
    mov wc.lpszMenuName,NULL
    mov wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov wc.hIcon,eax
    mov wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov wc.hCursor,eax
    invoke RegisterClassEx, addr wc

    invoke CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
    WS_OVERLAPPEDWINDOW,\
    CW_USEDEFAULT,CW_USEDEFAULT,258,160,NULL,NULL,\
    hInst,NULL
    mov ebx,eax
    invoke ShowWindow, ebx,SW_SHOWNORMAL
    invoke UpdateWindow, ebx


    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov eax,msg.wParam
    ret
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp

end start
```

Ora commentiamo le parti interessanti:

```
  invoke GetModuleHandle, NULL
  mov hInstance,eax
```

La chiamata a GetModuleHandle ci ritorna l'hInstance della nostra applicazione, che ci servirà per successive chiamate API.

```
  invoke GetCommandLine
  mov CommandLine,eax
```

GetCommandLine ci ritorna un puntatore ad una stringa contenente la riga di comando della nostra applicazione. In questo caso è totalmente inutile, ma l'ho inclusa perché ritenevo utile documentarla.

```
    mov wc.cbSize,SIZEOF WNDCLASSEX
    mov wc.style, CS_HREDRAW or CS_VREDRAW
    mov wc.lpfnWndProc, OFFSET WndProc
    mov wc.cbClsExtra,NULL
    mov wc.cbWndExtra,NULL
    push hInstance
    pop wc.hInstance
    mov wc.hbrBackground,COLOR_WINDOW
    mov wc.lpszMenuName,NULL
    mov wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov wc.hIcon,eax
    mov wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov wc.hCursor,eax
    invoke RegisterClassEx, offset wc
```

Questa parte registra la classe della finestra principale del nostro programma, dopo aver ovviamente riempito la struttura wc. LoadIcon serve a caricare un'icona, e in questo caso l'icona definita dalla costante IDI_APPLICATION. LoadCursor carica un cursore, e IDC_ARROW è il cursore standard, il classico cursore a forma di freccetta.

```
invoke CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
 WS_OVERLAPPEDWINDOW,\
CW_USEDEFAULT,CW_USEDEFAULT,258,160,NULL,NULL,\
hInst,NULL
mov ebx,eax
invoke ShowWindow, ebx,SW_SHOWNORMAL
invoke UpdateWindow, ebx
```

Dopo aver registrato la classe, la creiamo con una chiamata a CreateWindowEx. Lo stile passato è WS_OVERLAPPEDWINDOW. Questo stile crea una finestra con caption (barra del titolo), menu di sistema, ridimensionabile, con pulsanti di ingrandimento e riduzione ad icona. Dopo averla creata mettiamo temporaneamente in ebx il valore di eax, ovvero l'hwnd della finestra. In questo caso non ci serve salvarlo, ma lo utilizzeremo solo per qualche chiamata successiva. Perchè proprio in ebx? Perchè il valore del registro ebx non sarà MAI modificato da nessuna chiamata API. Le chiamate a ShowWindow ed UpdateWindow servono rispettivamente per visualizzare la finestra e costringerla ad un refresh, ovvero ridisegnarla. In realtà in questo caso avremmo potuto passare WS_VISIBLE come stile della CreateWindowEx e ci saremmo potuti risparmiare ShowWindow ed UpdateWindow.

```
    .WHILE TRUE
```

```
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov eax,msg.wParam
    ret
WinMain endp
```

Questo è il loop continuo di cui abbiamo già parlato. Notate ciò che facciamo all'uscita del loop: mettiamo in eax il valore di msg.wParam e ritorniamo (ret). Questo valore srà poi passato come parametro a ExitProcess. In realtà questo valore è ignorato da windows, ma la Microsoft ci dice di fare così, e quindi noi lo facciamo.

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
    .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp
```

Questa è la Window Procedure della nostra finestra. E' tramite questa procedura che noi possiamo interagire con l'utente, accorgerci se l'utente muove il mouse sulla nostra finestra, se clicca, se la sposta, ed essere notificati di qualunque altro evento degno di nota.
La window procedure di una finestra principale deve SEMPRE processare almeno il messaggio WM_DESTROY. Questo messaggio viene inviato non appena l'utente chiude la nostra finestra, e durante la gestione di questo messaggio noi dobbiamo chiamare la PostQuitMessage, che manderà alla funzione GetMessage il messaggio WM_QUIT, provocando un valore di ritorno uguale a 0, che a sua volta provocherà l'uscita dal loop e la chiusura del programma.
Da notare che quando la window procedure riceve il messaggio WM_DESTROY essa è già stata rimossa dallo schermo. Se vogliamo intercettare il momento in cui l'utente preme il pulsante di chiusura dobbiamo intercettare il messaggio WM_CLOSE, che ci permetterà di accertarci che l'utente voglia veramente chiudere, e in caso contrario annullare tutto.

Adesso facciamo un altro esempio, dove metteremo anche un pulsante nella finestra principale. :)

```
.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

WinMain proto :DWORD,:DWORD,:DWORD,:DWORD

.data
ClassName db "FirstWindow",0
AppName db "Wow! La nostra prima finestra!",0

BtnClass db "Button",0
BtnCaption db "Pulsante",0

MsgBoxTitle db "MessageBox",0
MsgBoxText db "Hai cliccato sul pulsante!",0

.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?

.code
start:
    invoke GetModuleHandle, NULL
    mov hInstance,eax
    invoke GetCommandLine
    mov CommandLine,eax
    invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
    invoke ExitProcess,eax

WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG

    mov wc.cbSize,SIZEOF WNDCLASSEX
    mov wc.style, CS_HREDRAW or CS_VREDRAW
    mov wc.lpfnWndProc, OFFSET WndProc
    mov wc.cbClsExtra,NULL
    mov wc.cbWndExtra,NULL
    push hInstance
    pop wc.hInstance
    mov wc.hbrBackground,COLOR_WINDOW
    mov wc.lpszMenuName,NULL
    mov wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov wc.hIcon,eax
    mov wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov wc.hCursor,eax
    invoke RegisterClassEx, addr wc

    invoke CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
    WS_OVERLAPPEDWINDOW,\
    CW_USEDEFAULT,CW_USEDEFAULT,260,160,NULL,NULL,\
    hInst,NULL
    mov ebx,eax

    invoke CreateWindowEx,NULL,ADDR BtnClass,ADDR BtnCaption,\
    BS_PUSHBUTTON OR WS_VISIBLE OR WS_CHILD,\
    47,25,150,70,ebx,NULL,\
    hInst,NULL
```

```
    invoke ShowWindow, ebx,SW_SHOWNORMAL
    invoke UpdateWindow, ebx


    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke TranslateMessage, ADDR msg
        invoke DispatchMessage, ADDR msg
    .ENDW
    mov eax,msg.wParam
    ret
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage, NULL
    .ELSEIF uMsg==WM_COMMAND
        invoke MessageBoxA, hWnd,offset MsgBoxText,offset MsgBoxTitle,NULL
    .ELSE
        invoke DefWindowProc, hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp

end start
```

Bene, stavolta commentiamo solo ciò che è nuovo:

```
invoke CreateWindowEx,NULL,ADDR ClassName,ADDR AppName,\
WS_OVERLAPPEDWINDOW,\
CW_USEDEFAULT,CW_USEDEFAULT,260,160,NULL,NULL,\
hInst,NULL
mov ebx,eax

invoke CreateWindowEx,NULL,ADDR BtnClass,ADDR BtnCaption,\
BS_PUSHBUTTON OR WS_VISIBLE OR WS_CHILD,\
47,25,150,70,ebx,NULL,\
hInst,NULL

invoke ShowWindow, ebx,SW_SHOWNORMAL
invoke UpdateWindow, ebx
```

La prima chiamata a CreateWindowEx crea la finestra principale. La seconda, invece crea il pulsante. Come classe stavolta passiamo "Button". Passiamo 3 stili: BS_PUSHBUTTON, che serve a dire a Windows di creare un PushButton (e non, ad esempio, una CheckBox); WS_VISIBLE, senza il quale il bottone non sarebbe visibile se non tramite una chiamata a ShowWindow, ma è sufficiente farlo con la finestra principale; WS_CHILD che è NECESSARIO con tutte le finestre che non sono figlie dirette del desktop, che altrimenti non funzionerebbero correttamente. Non salviamo il valore dell'handle del pulsante appena creato perché in questo caso non ci serve, dato che abbiamo un solo pulsante.

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
        invoke PostQuitMessage, NULL
    .ELSEIF uMsg==WM_COMMAND
        invoke MessageBoxA, hWnd,offset MsgBoxText,offset MsgBoxTitle,NULL
    .ELSE
        invoke DefWindowProc, hWnd,uMsg,wParam,lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp
```

Anche qui la nostra bella Window Procedure. Stavolta gestiamo anche il messaggio WM_COMMAND, che ci permette di intercettare il momento in cui viene cliccato il nostro pulsante. Se ci fossero stati più pulsanti, sarebbe stato necessario controllare qual è quello interessato tramite l'hwnd, che è contenuto in lParam, mentre wParam contiene un identificativo (ID), che in questo caso non ci interessa ma si renderà necessario conoscere quando utilizzeremo le dialog boxes.

Secondo approccio: le DialogBoxes

Il metodo della creazione di finestre tramite CreateWindowEx può andare bene nel caso di finestre contenenti pochi controlli. Tuttavia c'è un metodo più professionale e più efficace, e che in molti casi risulta più semplice: l'utilizzo delle Dialog Boxes, contenute nelle risorse delle applicazioni.

Per inserire una finestra di dialogo dobbiamo innanzitutto creare il file di risorse per l'applicazione. Esso può anche essere creato a mano con il notepad, ma è più semplice utilizzare un editor di risorse, come ad esempio il Borland Resource Workshop.
Le finestre di dialogo possono contenere al loro interno tutti i controlli che possono essere contenuti negli altri tipi di finestre. Ogni controllo possiede, oltre al normale hwnd, un identificativo (ID), che viene utilizzato in modo simile all'hwnd. Di solito le API che lavorano con i controlli delle Dialog Boxes utilizzano questo ID.
Per comunicare con le finestre figlie della DialogBox si può utilizzare l'API SendDlgItemMessage. La SendDialogItemMessage richiede 5 parametri:

```
LONG SendDlgItemMessage(

  HWND hDlg,      // handle of dialog box
  int nIDDlgItem,  // identifier of control
  UINT Msg,       // message to send
  WPARAM wParam,   // first message parameter
  LPARAM lParam    // second message parameter
);
```

hDlg - Identifica la Dialog Box che contiene il controllo.

nIDDlgItem - ID del controllo.

Msg - Messaggio da inviare.

wParam - Primo parametro del messaggio.

lParam - Secondo parametro del messaggio.

I messaggi destinati alla nostra Dialog Box vengono inviati ad una procedura chiamata Dialog Box Procedure, molto simile alle normali Window Procedure. Riceve esattamente gli stessi argomenti: *hDlg* (handle della DialogBox), *iMsg* (messaggio), *wParam* (Primo parametro), *lParam* (secondo parametro). L'unica differenza è che, come valore di ritorno, essa deve restituire TRUE o FALSE. TRUE è il valore da restituire nel caso in cui la funzione processa il messaggio. FALSE viene restituito in caso contrario.

Per utilizzare una DialogBox come finestra principale, si possono utilizzare due metodi. Il primo è quello di utilizzare la dialog per registrare una classe uguale utilizzando RegisterWindowEx. In questo modo si proseguirà poi alla solita maniera, con il loop e tutto il resto. Il secondo modo, più semplice, è quello di creare la DialogBox direttamente come finestra principale.

Ora analizzeremo un esempio di entrambi gli approcci. Questi esempi sono composti di 2 files, perché c'è anche il file di risorse.

```
=====================================================================
=                       Rsrc.rc                                     =
=====================================================================

#define DS_3DLOOK 4
#define DS_MODALFRAME 128
#define DS_CENTER 2048
#define WS_OVERLAPPED 0
#define WS_VISIBLE 268435456
#define WS_CAPTION 12582912
#define WS_SYSMENU 524288
#define WS_TABSTOP 65536
#define WS_BORDER 8388608
#define WS_VSCROLL 2097152

#define ES_MULTILINE 4
#define ES_READONLY 2048
#define ES_UPPERCASE 8


#define IDM_ESCI 1
#define IDM_FUNZ1 2
#define IDM_FUNZ2 3

#define IDC_BUTTON 100

MyDialog DIALOG 10, 10, 100, 60
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
CAPTION "La nostra prima Dialog!!!"
CLASS "DIALOGCLASS"
BEGIN
   DEFPUSHBUTTON "Button", IDC_BUTTON, 20,10,60,13
END

MyMenu MENU
{
 POPUP "&File"
       {
        MENUITEM "&Esci",IDM_ESCI
       }
 POPUP "F&unzioni"
       {
        MENUITEM "Funzione1",IDM_FUNZ1
        MENUITEM "Funzione2",IDM_FUNZ2
       }
}


=====================================================================


=====================================================================
=                       Dialog1.asm                                 =
=====================================================================

.386
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

include \masm32\M32LIB\masm32.inc
includelib \masm32\M32LIB\masm32.lib


WinMain        proto :DWORD,:DWORD,:DWORD,:DWORD

.const
IDM_ESCI     equ 1
IDM_FUNZ1    equ 2
IDM_FUNZ2    equ 3

IDC_BUTTON   equ 100

.data
ClassName       db "DIALOGCLASS",0
DlgName         db "MyDialog",0
AppName         db "La nostra prima Dialog!",0
MenuName        db "MyMenu",0

MsgBoxTitle     db "Click!",0

MsgBoxFunz1     db "Hai cliccato su Funzione1!!",0
MsgBoxFunz2     db "Hai cliccato su Funzione2!!",0

MsgBoxButton    db "Hai cliccato sul pulsante!!",0
```

```
.data?
hInstance   HINSTANCE   ?
CommandLine LPSTR       ?

.code
start:
      invoke GetModuleHandle, NULL
      mov    hInstance,eax
      invoke GetCommandLine
      mov    CommandLine,eax
      invoke WinMain, hInstance,NULL,CommandLine, SW_SHOWDEFAULT
      invoke ExitProcess,eax

WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
    LOCAL wc:WNDCLASSEX
    LOCAL msg:MSG
    LOCAL hDlg:HWND
    mov   wc.cbSize,SIZEOF WNDCLASSEX
    mov   wc.style, CS_HREDRAW or CS_VREDRAW
    mov   wc.lpfnWndProc, OFFSET WndProc
    mov   wc.cbClsExtra,NULL
    mov   wc.cbWndExtra,DLGWINDOWEXTRA
    push  hInst
    pop   wc.hInstance
    mov   wc.hbrBackground,COLOR_BTNFACE+1
    mov   wc.lpszMenuName,OFFSET MenuName
    mov   wc.lpszClassName,OFFSET ClassName
    invoke LoadIcon,NULL,IDI_APPLICATION
    mov   wc.hIcon,eax
    mov   wc.hIconSm,eax
    invoke LoadCursor,NULL,IDC_ARROW
    mov   wc.hCursor,eax
    invoke RegisterClassEx, addr wc
    invoke CreateDialogParam,hInstance,ADDR DlgName,NULL,NULL,NULL
    mov   hDlg,eax
    invoke ShowWindow, hDlg,SW_SHOWNORMAL
    invoke UpdateWindow, hDlg

    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke IsDialogMessage, hDlg, ADDR msg
        .IF eax ==FALSE
            invoke TranslateMessage, ADDR msg
            invoke DispatchMessage, ADDR msg
        .ENDIF
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_DESTROY
          invoke PostQuitMessage,NULL
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF ax == IDM_ESCI
            invoke SendMessageA, hWnd,WM_CLOSE,NULL,NULL
        .ELSEIF ax == IDM_FUNZ1
            invoke MessageBoxA, hWnd,offset MsgBoxFunz1,offset MsgBoxTitle, NULL
        .ELSEIF ax == IDM_FUNZ2
            invoke MessageBoxA, hWnd,offset MsgBoxFunz2,offset MsgBoxTitle, NULL
        .ELSEIF ax == IDC_BUTTON
            invoke MessageBoxA, hWnd,offset MsgBoxButton,offset MsgBoxTitle, NULL
        .ENDIF
    .ELSE
          invoke DefWindowProc,hWnd,uMsg,wParam,lParam
          ret
     .ENDIF
     xor eax,eax
     ret
WndProc endp

end start
```

E ora le solite spiegazioni:

```
#define DS_3DLOOK 4
#define DS_MODALFRAME 128
#define DS_CENTER 2048
#define WS_OVERLAPPED 0
#define WS_VISIBLE 268435456
#define WS_CAPTION 12582912
#define WS_SYSMENU 524288
#define WS_TABSTOP 65536
#define WS_BORDER 8388608
#define WS_VSCROLL 2097152

#define ES_MULTILINE 4
#define ES_READONLY 2048
#define ES_UPPERCASE 8


#define IDM_ESCI 1
#define IDM_FUNZ1 2
#define IDM_FUNZ2 3

#define IDC_BUTTON 100
```

```
MyDialog DIALOG 10, 10, 100, 60
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
CAPTION "La nostra prima Dialog!!!"
CLASS "DIALOGCLASS"
BEGIN
   DEFPUSHBUTTON "Button", IDC_BUTTON, 20,10,60,13
END

MyMenu MENU
{
 POPUP "&File"
       {
        MENUITEM "&Esci",IDM_ESCI
       }
 POPUP "F&unzioni"
       {
        MENUITEM "Funzione1",IDM_FUNZ1
        MENUITEM "Funzione2",IDM_FUNZ2
       }
}
```

4
Questo codice è stato generato da un editor di risorse e successivamente modificato un po' a mano. Io uso il Borland Resource Workshop, ma in giro ne trovate molti altri, come quello incluso nel VC++. E' importante la parola chiave CLASS seguita dal nome della classe che vogliamo creare. Senza di essa, infatti, non potremmo utilizzare questo sistema.

```
.const
IDM_ESCI     equ 1
IDM_FUNZ1    equ 2
IDM_FUNZ2    equ 3

IDC_BUTTON   equ 100
```

Queste costanti sono le stesse inserite nello script di risorse, e ci possono servire per molteplici scopi (ad esempio individuare i click nei menu o nei pulsanti).

```
.data
ClassName       db "DIALOGCLASS",0
DlgName         db "MyDialog",0
AppName         db "La nostra prima Dialog!",0
MenuName        db "MyMenu",0
```

Lo scopo di queste stringhe è intuibile dal nome, giusto? :-)

```
WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,CmdLine:LPSTR,CmdShow:DWORD
      LOCAL wc:WNDCLASSEX
      LOCAL msg:MSG
      LOCAL hDlg:HWND
      mov   wc.cbSize,SIZEOF WNDCLASSEX
      mov   wc.style, CS_HREDRAW or CS_VREDRAW
      mov   wc.lpfnWndProc, OFFSET WndProc
      mov   wc.cbClsExtra,NULL
      mov   wc.cbWndExtra,DLGWINDOWEXTRA
      push  hInst
      pop   wc.hInstance
      mov   wc.hbrBackground,COLOR_BTNFACE+1
      mov   wc.lpszMenuName,OFFSET MenuName
      mov   wc.lpszClassName,OFFSET ClassName
      invoke LoadIcon,NULL,IDI_APPLICATION
      mov   wc.hIcon,eax
      mov   wc.hIconSm,eax
      invoke LoadCursor,NULL,IDC_ARROW
      mov   wc.hCursor,eax
      invoke RegisterClassEx, addr wc
```

Questa parte registra una classe identica alla dialog del nostro script. L'unica differenza rispetto agli esempi precedenti è che il membro *cbWndExtra* viene settato al valore di DLGWINDOWEXTRA, mentre precedentemente avevamo messo NULL. Notate che come ClassName passiamo il nome della CLASS usato nello script di risorse.

```
      invoke CreateDialogParam,hInstance,ADDR DlgName,NULL,NULL,NULL
      mov   hDlg,eax
```

Stavolta per creare la finestra utilizziamo CreateDialogParam invece di CreateWindowEx. L'API CreateDialogParam serve appunto a creare una Dialog basata su un template contenuto nelle risorse di un programma. Vediamo il prototipo della funzione:

```
HWND CreateDialogParam(

  HINSTANCE hInstance,     // handle to application instance
  LPCTSTR lpTemplateName,  // identifies dialog box template
  HWND hWndParent,         // handle to owner window
  DLGPROC lpDialogFunc,    // pointer to dialog box procedure
  LPARAM dwInitParam       // initialization value
);
```

hInstance - Questo lo conosciamo già: è l'handle dell'istanza della nostra applicazione.

lpTemplateName - In questo parametro dobbiamo passare il nome della Dialog, lo stesso nome usato nello script di risorse. Da non confondere con la keyword CLASS!

hWndParent - Handle della finestra genitrice. Generalmente NULL.

lpDialogFung - Puntatore alla window procedure della dialog.

dwInitParam - Valore passato al parametro *lParam* nel messaggio WM_INITDIALOG. Questo messaggio viene mandato prima della visualizzazione della finestra.

In questo caso per gli ultimi 3 parametri passiamo NULL. Passando NULL ad *hWndParent* assumiamo come parent la finestra Desktop. Passiamo NULL anche per *lpDialogFunc* perchè la window procedure l'abbiamo già indicata tramite RegisterClassEx. *dwInitParam* invece in questo caso non ci serve, e avremmo potuto passare qualunque altro valore.

```
      invoke ShowWindow, hDlg,SW_SHOWNORMAL
```

```
    invoke UpdateWindow, hDlg
```

Come al solito visualizziamo e facciamo il refresh della finestra.

```
    .WHILE TRUE
        invoke GetMessage, ADDR msg,NULL,0,0
        .BREAK .IF (!eax)
        invoke IsDialogMessage, hDlg, ADDR msg
        .IF eax == FALSE
            invoke TranslateMessage, ADDR msg
            invoke DispatchMessage, ADDR msg
        .ENDIF
    .ENDW
    mov     eax,msg.wParam
    ret
WinMain endp
```

Stavolta c'è una piccola modifica nel loop. Richiamiamo IsDialogMessage, che si occupa di processare i messaggi destinati alle dialogs. Quando il messaggio non viene processato restituisce FALSE, e quindi ci occupiamo del messaggio chiamando TranslateMessage e DispatchMessage.

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
      .IF uMsg==WM_DESTROY
        invoke PostQuitMessage,NULL
      .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF ax == IDM_ESCI
            invoke SendMessageA, hWnd,WM_CLOSE,NULL,NULL
        .ELSEIF ax == IDM_FUNZ1
            invoke MessageBoxA, hWnd,offset MsgBoxFunz1,offset MsgBoxTitle, NULL
        .ELSEIF ax == IDM_FUNZ2
            invoke MessageBoxA, hWnd,offset MsgBoxFunz2,offset MsgBoxTitle, NULL
        .ELSEIF ax == IDC_BUTTON
            invoke MessageBoxA, hWnd,offset MsgBoxButton,offset MsgBoxTitle, NULL
        .ENDIF
      .ELSE
        invoke DefWindowProc,hWnd,uMsg,wParam,lParam
        ret
      .ENDIF
      xor eax,eax
      ret
WndProc endp
```

E come al solito la Window Procedure, in cui proseguiamo come negli esempi precedenti.

Adesso esaminiamo il secondo approccio. Con questo sistema non sono più necessari il message-loop né la registrazione della class con RegisterClassEx.

```
=====================================================================
=                       Rsrc.rc                                     =
=====================================================================

#define DS_3DLOOK 4
#define DS_MODALFRAME 128
#define DS_CENTER 2048
#define WS_OVERLAPPED 0
#define WS_VISIBLE 268435456
#define WS_CAPTION 12582912
#define WS_SYSMENU 524288
#define WS_TABSTOP 65536
#define WS_BORDER 8388608
#define WS_VSCROLL 2097152



#define ES_MULTILINE 4
#define ES_READONLY 2048
#define ES_UPPERCASE 8


#define IDM_ESCI 1
#define IDM_FUNZ1 2
#define IDM_FUNZ2 3

#define IDC_BUTTON 100

MyDialog DIALOG 10, 10, 100, 50
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
CAPTION "La nostra prima Dialog!!!"
MENU MyMenu
BEGIN
   DEFPUSHBUTTON   "Button", IDC_BUTTON,    20,10,60,13
END

MyMenu MENU
{
 POPUP "&File"
       {
        MENUITEM "&Esci",IDM_ESCI
       }
 POPUP "F&unzioni"
       {
        MENUITEM "Funzione1",IDM_FUNZ1
        MENUITEM "Funzione2",IDM_FUNZ2
       }
}

=====================================================================
```

```
;=====================================================================
;=                    Dialog2.asm                                    =
;=====================================================================

.386
.model flat,stdcall

option casemap:none

DlgProc proto :DWORD,:DWORD,:DWORD,:DWORD

include \masm32\include\windows.inc
include \masm32\include\user32.inc
include \masm32\include\kernel32.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib

.data

DlgName db "MyDialog",0
AppName db "La nostra prima dialog!!",0

MsgBoxTitle db "Click!",0

MsgBoxFunz1 db "Hai cliccato su Funzione1!!",0
MsgBoxFunz2 db "Hai cliccato su Funzione2!!",0

MsgBoxButton db "Hai cliccato sul pulsante!!",0


.data?
hInstance HINSTANCE ?
CommandLine LPSTR ?

.const

IDM_ESCI equ 1
IDM_FUNZ1 equ 2
IDM_FUNZ2 equ 3

IDC_BUTTON equ 100

.code
start:
	invoke GetModuleHandle, NULL
	mov hInstance,eax
	invoke DialogBoxParam, hInstance, ADDR DlgName,NULL, addr DlgProc, NULL
	invoke ExitProcess,eax

DlgProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
	.IF uMsg==WM_CLOSE
		invoke SendMessage,hWnd,WM_COMMAND,IDM_ESCI,0
	.ELSEIF uMsg==WM_COMMAND
		mov eax,wParam
		.IF lParam==0
			.IF ax==IDM_ESCI
				invoke EndDialog, hWnd,NULL
			.ELSEIF ax==IDM_FUNZ1
				invoke MessageBoxA, hWnd, OFFSET MsgBoxFunz1,OFFSET MsgBoxTitle, MB_ICONINFORMATION
			.ELSEIF ax==IDM_FUNZ2
				invoke MessageBoxA, hWnd, OFFSET MsgBoxFunz2,OFFSET MsgBoxTitle, MB_ICONINFORMATION
			.ENDIF
		.ELSE
			mov edx,wParam
			shr edx,16
			.IF dx==BN_CLICKED
				.IF ax==IDC_BUTTON
					invoke MessageBoxA, hWnd, OFFSET MsgBoxButton,OFFSET MsgBoxTitle, MB_ICONINFORMATION
				.ENDIF
			.ENDIF
		.ENDIF
	.ELSE
		mov eax,FALSE
		ret
	.ENDIF
	mov eax,TRUE
	ret
DlgProc endp

end start

;=====================================================================
```

Analizziamo quest'ultimo esempio:

```
MyDialog DIALOG 10, 10, 100, 50
STYLE 0x0004 | DS_CENTER | WS_CAPTION | WS_SYSMENU | WS_VISIBLE | WS_OVERLAPPED | DS_MODALFRAME | DS_3DLOOK
CAPTION "La nostra prima Dialog!!!"
MENU MyMenu
BEGIN
   DEFPUSHBUTTON   "Button", IDC_BUTTON,    20,10,60,13
END
```

Stavolta non abbiamo bisogno della parola chiave CLASS. Notate invece che utilizziamo la parola chiave MENU, che ci permette di settare automaticamente il menu della dialog, cosa che altrimenti dovremmo fare a mano.

```
start:
	invoke GetModuleHandle, NULL
```

```
    mov hInstance,eax
    invoke DialogBoxParam, hInstance, ADDR DlgName,NULL, addr DlgProc, NULL
    invoke ExitProcess,eax
```

Stavolta non abbiamo bisogno di né di WinMain né di message-loop. Semplicemente richiamiamo l'API DialogBoxParam che provvederà a tutto il resto :-)

```
DlgProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM, lParam:LPARAM
    .IF uMsg==WM_CLOSE
        invoke SendMessage,hWnd,WM_COMMAND,IDM_ESCI,0
```

Se vogliamo utilizzare questo sistema, dobbiamo anche processare il messaggio WM_CLOSE, in quanto il sistema operativo non si occupa di mandare il messaggio WM_DESTROY quando processa un WM_CLOSE. Se non processassimo WM_CLOSE quando l'utente cliccherebbe sulla X di chiusura finestra non succederebbe niente. In questo caso utilizziamo SendMessage per simulare un clic nel sottomenu "Esci". Sarebbe stata la stessa cosa richiamare EndDialog da qui.

```
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF lParam==0
            .IF ax==IDM_ESCI
                invoke EndDialog, hWnd,NULL
            .ELSEIF ax==IDM_FUNZ1
                invoke MessageBoxA, hWnd, OFFSET MsgBoxFunz1,OFFSET MsgBoxTitle, MB_ICONINFORMATION
            .ELSEIF ax==IDM_FUNZ2
                invoke MessageBoxA, hWnd, OFFSET MsgBoxFunz2,OFFSET MsgBoxTitle, MB_ICONINFORMATION
            .ENDIF
        .ELSE
            mov edx,wParam
            shr edx,16
            .IF dx==BN_CLICKED
                .IF ax==IDC_BUTTON
                    invoke MessageBoxA, hWnd, OFFSET MsgBoxButton,OFFSET MsgBoxTitle, MB_ICONINFORMATION
                .ENDIF
            .ENDIF
        .ENDIF
    .ELSE
        mov eax,FALSE
        ret
    .ENDIF
    mov eax,TRUE
    ret
DlgProc endp
```

Il resto della Dialog Procedure è uguale al solito. Quando clickiamo sul menu Esci viene richiamata l'API EndDialog, che si occupa di terminare la nostra dialog. EndDialog richiede 2 parametri: il primo è l'handle della dialog; il secondo è il valore di ritorno che avrà l'API che abbiamo utilizzato per creare la DialogBox (in questo caso DialogBoxParam).

E con questo si chiude questa piccola guida sulla creazione di finestre in Windows.

Spider

Note finali

Ringrazio Albe che mi ha suggerito di scrivere questo tute e Iczelion i cui tute sono sempre un'ottima documentazione :)

Disclaimer

Home