LECTURE-2

A Simple Window

In this lecture we will study a program, how to make a simple window ? Well it's not entirely that simple I'm afraid. It's not difficult once you know what you're doing but there are quite a few things you need to do to get a window to show up.

It is always better to do things first and learn them later, so here is the code to a simple window which will be explained shortly. Before we write a program, we will like to discuss about few definitions to be used in the program.

2.1 WINDOW 98 DEFINES SEVERAL STRUCTURES.

Apart from the datatype discussed in lesson -1, windows define structures:

- MSG-Holds window 98 message.
- WNDCLASS- defines a window class.
- WNDCLASSEX;- defines a window extended class with two extra members.

2.2 WINDOW MESSAGES

The **events** are the user action or the occurrence that the windows handle. For example clicking at the mouse, dragging a window, pressing a key or drawing a pictures are the events that window record. These events act as messages for the window and each message is placed in a message queue for further processing.   For handling these messages, window program provides a function **wndproc()** or **windowproc(). Windows** communicate with the program through this function. It receives the messages by means of arguments.

2.3 PROGRAM TO CREATE A WINDOW.

```
 #include <windows.h>

const char g_szClassName[] = "myWindowClass";

// Step 4: the Window Procedure
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam)
{
    switch(msg)
    {
        case WM_CLOSE:
            DestroyWindow(hwnd);
        break;
        case WM_DESTROY:
            PostQuitMessage(0);
        break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG msg;

    //Step 1: Registering the Window Class
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
            MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    // Step 2: Creating the Window
    hwnd = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        g_szClassName,
        "The title of my window",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
        NULL, NULL, hInstance, NULL);

    if(hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
            MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);
    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```

For most part this is the simplest windows program you can write that actually creates a functional window, a mere 70 or so lines. If you got the first example to compile then this one should work with no problems.

Step 1: Registering the Window Class

A *Window Class* stores information about a type of window, including it's *Window Procedure* which controls the window, the small and large icons for the window, and the background color. This way, you can register a class once, and create as many windows as you want from it, without having to specify all those attributes over and over. Most of the attributes you set in the window class can be changed on a per-window basis if desired.

A Window Class has NOTHING to do with C++ classes.

```
const char g_szClassName[] = "myWindowClass";
```

The variable above stores the name of our window class, we will use it shortly to register our window class with the system.

```
    WNDCLASSEX wc;
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
            MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }
```

Members

cbSize

Specifies the size, in bytes, of this structure. Set this member to sizeof(WNDCLASSEX). Be sure to set this member before calling the **GetClassInfoEx** function.

style

Specifies the class style(s). Styles can be combined by using the bitwise OR (|) operator. This member can be any combination of the following values:

Value	**Action**
CS_BYTEALIGNCLIENT	Aligns the window's client area on the byte boundary (in the x direction). This style affects the width of the window and its horizontal position on the display.
CS_BYTEALIGNWINDOW	Aligns a window on a byte boundary (in the x direction). This style affects the width of the window and its horizontal position on the display.
CS_CLASSDC	Allocates one device context to be shared by all windows in the class. Because window classes are process specific, it is possible for multiple threads of an application to create a window of the same class. It is also possible for the threads to attempt to use the device context simultaneously. When this happens, the system allows only one thread to successfully finish its drawing operation. For more information, see Device Contexts.
CS_DBLCLKS	Sends double-click messages to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class.
CS_GLOBALCLASS	Allows an application to create a window of the class regardless of the value of the *hInstance* parameter passed to the **CreateWindowEx** function. If you do not specify this style, the *hInstance* parameter passed to the **CreateWindow** (or **CreateWindowEx**) function must be the same as the *hInstance* parameter passed to the **RegisterClassEx** function.

	You can create a global class by creating the window class in a dynamic-link library (DLL) and listing the name of the DLL in the registry under the following keys: **HKEY_LOCAL_MACHINE\Software \Microsoft\Windows NT\ CurrentVersion\Windows\AppInit_DLLs** Whenever a process starts, the system loads the specified DLLs in the context of the newly started process before calling the entry-point function in that process. The DLL must register the class during its initialization procedure and must specify the CS_GLOBALCLASS style.
CS_HREDRAW	Redraws the entire window if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE	Disables **Close** on the **window** menu.
CS_OWNDC	Allocates a unique device context window in the class. For more information, see Device Contexts.
CS_PARENTDC	Sets the clipping region of the child window to that of the parent window so that the child can draw on the parent. A window with the CS_PARENTDC style bit receives a regular device context from the system's cache of device contexts. It does not give the child the parent's device context or device context settings. Specifying CS_PARENTDC enhances an application's performance. For more information, see Device Contexts.
CS_SAVEBITS	Saves, as a bitmap, the portion of the screen image obscured by a window. The system uses the saved bitmap to re-create the screen image when the window is removed. The system displays the bitmap at its original location and does not send WM_PAINT messages to windows obscured by the window if the memory used by the bitmap has not been discarded and if other screen actions have not invalidated the stored image. This style is useful for small windows (for example, menus or dialog boxes) that are displayed briefly and then removed before other screen activity takes place. This style increases the time required to

	display the window, because the system must first allocate memory to store the bitmap.
CS_VREDRAW	Redraws the entire window if a movement or size adjustment changes the height of the client area.

lpfnWndProc
Pointer to the window procedure. You must use the **CallWindowProc** function to call the window procedure. For more information, see **WindowProc**.

cbClsExtra
Specifies the number of extra bytes to allocate following the window-class structure. The system initializes the bytes to zero.

cbWndExtra
Specifies the number of extra bytes to allocate following the window instance. The system initializes the bytes to zero. If an application uses **WNDCLASSEX** to register a dialog box created by using the **CLASS** directive in the resource file, it must set this member to DLGWINDOWEXTRA.

hInstance
Handle to the instance that the window procedure of this class is within.

hIcon
Handle to the class icon. This member must be a handle of an icon resource. If this member is NULL, an application must draw an icon whenever the user minimizes the application's window.

hCursor
Handle to the class cursor. This member must be a handle of a cursor resource. If this member is NULL, an application must explicitly set the cursor shape whenever the mouse moves into the application's window.

hbrBackground
Handle to the class background brush. This member can be a handle to the physical brush to be used for painting the background, or it can be a color value. A color value must be one of the following standard system colors (the value 1 must be added to the chosen color). If a color value is given, you must convert it to one of the following **HBRUSH** types:

COLOR_ACTIVEBORDER
COLOR_ACTIVECAPTION
COLOR_APPWORKSPACE
COLOR_BACKGROUND
COLOR_BTNFACE
COLOR_BTNSHADOW
COLOR_BTNTEXT
COLOR_CAPTIONTEXT
COLOR_GRAYTEXT
COLOR_HIGHLIGHT
COLOR_HIGHLIGHTTEXT
COLOR_INACTIVEBORDER

COLOR_INACTIVECAPTION
COLOR_MENU
COLOR_MENUTEXT
COLOR_SCROLLBAR
COLOR_WINDOW
COLOR_WINDOWFRAME
COLOR_WINDOWTEXT

The system automatically deletes class background brushes when the class is freed. An application should not delete these brushes, because a class may be used by multiple instances of an application.

When this member is NULL, an application must paint its own background whenever it is requested to paint in its client area. To determine whether the background must be painted, an application can either process the WM_ERASEBKGND message or test the **fErase** member of the **PAINTSTRUCT** structure filled by the **BeginPaint** function.

lpszMenuName

Pointer to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file. If you use an integer to identify the menu, use the **MAKEINTRESOURCE** macro. If this member is NULL, windows belonging to this class have no default menu.

lpszClassName

Pointer to a null-terminated string or is an atom. If this parameter is an atom, it must be a global atom created by a previous call to the **GlobalAddAtom** function. The atom, a 16-bit value, must be in the low-order word of **lpszClassName**; the high-order word must be zero.

If **lpszClassName** is a string, it specifies the window class name.

hIconSm

Handle to a small icon that is associated with the window class. If this member is NULL, the system searches the icon resource specified by the **hIcon** member for an icon of the appropriate size to use as the small icon.

Don't worry if that doesn't make much sense to you yet, the various parts that count will be explained more later. Another thing to remember Is, to not try and remember this stuff. I rarely (never) memorize structs, or function parameters, this is a waste of effort and, more importantly, time. If you know the functions you need to call then it is a matter of seconds to look up the exact parameters in your help files. If you don't have help files, get them. You are lost without. Eventually you will come to know the parameters to the functions you use most.

We then call RegisterClassEx() and check for failure, if it fails we pop up a message which says so and abort the program by returning from the WinMain() function.

Step 2: Creating the Window

Once the class is registered, we can create a window with it. You should look up the paramters for CreateWindowEx() (as you should ALWAYS do when using a new API call)., Prototype of CreatWindowEX() is as:

```
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
);
```

which is written in program as:

```
    HWND hwnd;
    hwnd = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        g_szClassName,
        "The title of my window",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
        NULL, NULL, hInstance, NULL);
```

WS_EX_CLIENTEDGE : is the extended windows style, WS which gives a sunken inner border around the window. Set it to 0 if you'd like to see the difference. Also play with other values to see what they do.

Next we have the class name (g_szClassName), this tells the system what kind of window to create. Since we want to create a window from the class we just registered, we use the name of that class. After that we specify our window name or title which is the text that will be displayed in the *Caption*, or *Title Bar* on our window.

The parameter we have as WS_OVERLAPPEDWINDOW is the *Window Style* parameter. There are quite a few of these and you should look them up and experiment to find out what they do. These will be covered more later.

The next four parameters (CW_USEDEFAULT, CW_USEDEFAULT, 320, 240) are the X and Y co-ordinates for the top left corner of your window, and the width and height of the window. I've set the X and Y values to CW_USEDEFAULT to let windows choose where on the screen to put the window. Remember that the left of the screen is an X value of zero and it increases to the right; The top of the screen is a Y value of zero which increases towards the bottom. The units are pixels, which is the smallest unit a screen can display at a given resolution.

Next (NULL, NULL, g_hInst, NULL) we have the ***Parent Window* handle**, the **menu handle**, the **application instance handle**, and a **pointer to window** creation data. In windows, the windows on your screen are arranged in a hierarchy of parent and child windows. When you see a button on a window, the button is the *Child* and it is contained within the window that is it's *Parent*. In this example, the parent handle is NULL because we have no parent, this is our main or *Top Level* window. The menu is NULL for now since we don't have one yet. The instance handle is set to the value that is passed in as the first parameter to WinMain(). The creation data that can be used to send additional data to the window that is being created is also NULL.

If you're wondering what this magic NULL is, it's simply defined as 0 (zero). Actually, in C it's defined as ((void*)0), since it's intended for use with pointers. Therefore you will possibly get warnings if you use NULL for integer values, depending on your compiler and the warning level settings. You can choose to ignore the warnings, or just use 0 instead.

Number one cause of people not knowing what the heck is wrong with their programs is probably that they didn't check the return values of their calls to see if they failed or not. CreateWindow() *will* fail at some point even if you're an experianced coder, simply because there are lots of mistakes that are easy to make. Untill you learn how to quickly identify those mistakes, at least give yourself the chance of figuring out where things go wrong, and **Always check return values!**

```
   if(hwnd == NULL)
   {
      MessageBox(NULL, "Window Creation Failed!", "Error!",
         MB_ICONEXCLAMATION | MB_OK);
      return 0;
   }
```

After we've created the window and checked to make sure we have a valid handle we show the window, using the last parameter in WinMain() and then update it to ensure that it has properly redrawn itself on the screen.

```
  ShowWindow(hwnd, nCmdShow);
  UpdateWindow(hwnd);
```

The nCmdShow parameter is optional, you could simply pass in SW_SHOWNORMAL all the time and be done with it. However using the parameter passed into WinMain() gives whoever is running your program to specify whether or not they want your window to start off visible, maximized, minimized, etc... You will find options for these in the properties of windows shortcuts, and this parameter is how the choice is carried out.

Step 3: The Message Loop

This is the heart of the whole program, pretty much everything that your program does passes through this point of control.

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

GetMessage() gets a message from your application's message queue. Any time the user moves the mouse, types on the keyboard, clicks on your window's menu, or does any number of other things, messages are generated by the system and entered into your program's message queue. By calling GetMessage() you are requesting the next available message to be removed from the queue and returned to you for processing. If there is no message, GetMessage() *Blocks*. If you are unfamiliar with the term, it means that it waits untill there is a message, and then returns it to you.

TranslateMessage() does some additional processing on keyboard events like generating WM_CHAR messages to go along with WM_KEYDOWN messages. Finally DispatchMessage() sends the message out to the window that the message was sent to. This could be our main window or it could be another one, or a control, and in some cases a window that was created behind the scenes by the system or another program. This isn't something you need to worry about because all we are concerned with is that we get the message and send it out, the system takes care of the rest making sure it gets to the proper window.

Step 4: the Window Procedure

If the message loop is the heart of the program, the window procedure is the brain. This is where all the messages that are sent to our window get processed.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam,
LPARAM lParam)
{
   switch(msg)
   {
      case WM_CLOSE:
         DestroyWindow(hwnd);
```

```
        break;
        case WM_DESTROY:
            PostQuitMessage(0);
        break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}
```

The window procedure is called for each message, the HWND parameter is the handle of your window, the one that the message applies to. This is important since you might have two or more windows of the same class and they will use the same window procedure (WndProc()). The difference is that the parameter hwnd will be different depending on which window it is. For example when we get the WM_CLOSE message we destroy the window. Since we use the window handle that we received as the first paramter, any other windows will not be affected, only the one that the message was intended for.

WM_CLOSE is sent when the user presses the Close Button or types Alt-F4. This will cause the window to be destroyed by default, but I like to handle it explicitly, since this is the perfect spot to do cleanup checks, or ask the user to save files etc. before exiting the program.

When we call DestroyWindow() the system sends the WM_DESTROY message to the window getting destroyed, in this case it's our window, and then destroys any remaining child windows before finally removing our window from the system. Since this is the only window in our program, we are all done and we want the program to exit, so we call PostQuitMessage(). This posts the WM_QUIT message to the message loop. We never receive this message, because it causes GetMessage() to return FALSE, and as you'll see in our message loop code, when that happens we stop processing messages and return the final result code, the wParam of WM_QUIT which happens to be the value we passed into PostQuitMessage(). The return value is only really useful if your program is designed to be called by another program and you want to return a specific value.

Step 5: There is no Step 5

Well that's it! If stuff haven't been explained clearly enough yet, just hang in there and hopefully things will become more clear as we get into more useful programs.