Managing Application State

Article • 08/19/2020

A window procedure is just a function that gets invoked for every message, so it is inherently stateless. Therefore, you need a way to track the state of your application from one function call to the next.

The simplest approach is simply to put everything in global variables. This works well enough for small programs, and many of the SDK samples use this approach. In a large program, however, it leads to a proliferation of global variables. Also, you might have several windows, each with its own window procedure. Keeping track of which window should access which variables becomes confusing and error-prone.

The CreateWindowEx function provides a way to pass any data structure to a window. When this function is called, it sends the following two messages to your window procedure:

- WM_NCCREATE
- WM_CREATE

These messages are sent in the order listed. (These are not the only two messages sent during CreateWindowEx, but we can ignore the others for this discussion.)

The WM_NCCREATE and WM_CREATE message are sent before the window becomes visible. That makes them a good place to initialize your UI—for example, to determine the initial layout of the window.

The last parameter of CreateWindowEx is a pointer of type **void***. You can pass any pointer value that you want in this parameter. When the window procedure handles the WM_NCCREATE or WM_CREATE message, it can extract this value from the message data.

Let's see how you would use this parameter to pass application data to your window. First, define a class or structure that holds state information.

C++

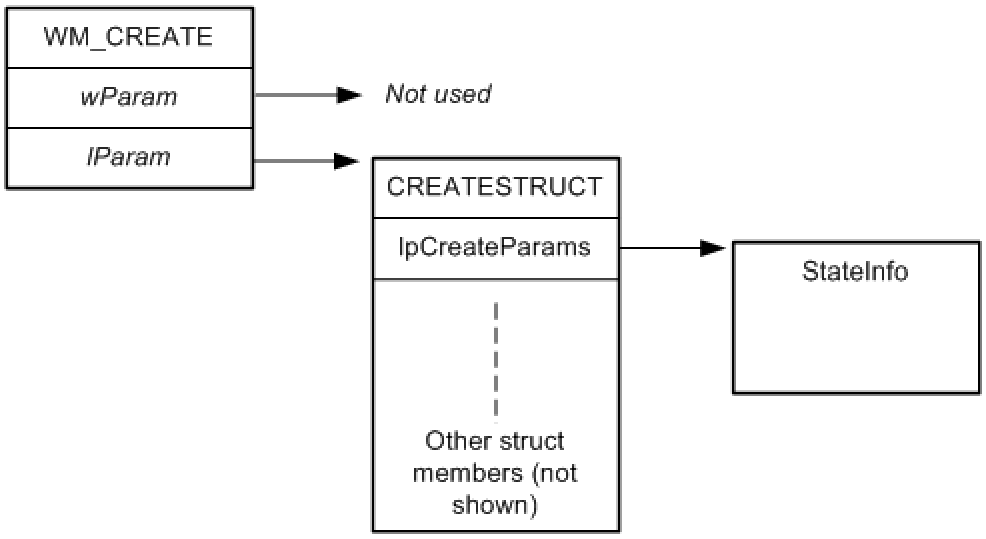
```cpp
// Define a structure to hold some state information.

struct StateInfo {
    // ... (struct members not shown)
};
```

When you call CreateWindowEx, pass a pointer to this structure in the final **void*** parameter.

C++

```
StateInfo *pState = new (std::nothrow) StateInfo;

if (pState == NULL)
{
    return 0;
}

// Initialize the structure members (not shown).

HWND hwnd = CreateWindowEx(
    0,                              // Optional window styles.
    CLASS_NAME,                     // Window class
    L"Learn to Program Windows",    // Window text
    WS_OVERLAPPEDWINDOW,            // Window style

    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL,       // Parent window
    NULL,       // Menu
    hInstance,  // Instance handle
    pState      // Additional application data
    );
```

When you receive the WM_NCCREATE and WM_CREATE messages, the *lParam* parameter of each message is a pointer to a CREATESTRUCT structure. The **CREATESTRUCT** structure, in turn, contains the pointer that you passed into CreateWindowEx.



Here is how you extract the pointer to your data structure. First, get the CREATESTRUCT structure by casting the *lParam* parameter.

C++

```
CREATESTRUCT *pCreate = reinterpret_cast<CREATESTRUCT*>(lParam);
```

The **lpCreateParams** member of the CREATESTRUCT structure is the original void pointer that you specified in CreateWindowEx. Get a pointer to your own data structure by casting **lpCreateParams**.

C++

```
pState = reinterpret_cast<StateInfo*>(pCreate->lpCreateParams);
```

Next, call the SetWindowLongPtr function and pass in the pointer to your data structure.

C++

```
SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pState);
```

The purpose of this last function call is to store the *StateInfo* pointer in the instance data for the window. Once you do this, you can always get the pointer back from the window by calling GetWindowLongPtr:

C++

```
LONG_PTR ptr = GetWindowLongPtr(hwnd, GWLP_USERDATA);
StateInfo *pState = reinterpret_cast<StateInfo*>(ptr);
```

Each window has its own instance data, so you can create multiple windows and give each window its own instance of the data structure. This approach is especially useful if you define a class of windows and create more than one window of that class—for example, if you create a custom control class. It is convenient to wrap the GetWindowLongPtr call in a small helper function.

C++

```
inline StateInfo* GetAppState(HWND hwnd)
{
    LONG_PTR ptr = GetWindowLongPtr(hwnd, GWLP_USERDATA);
    StateInfo *pState = reinterpret_cast<StateInfo*>(ptr);
    return pState;
}
```

Now you can write your window procedure as follows.

C++

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
    StateInfo *pState;
    if (uMsg == WM_CREATE)
    {
        CREATESTRUCT *pCreate = reinterpret_cast<CREATESTRUCT*>(lParam);
        pState = reinterpret_cast<StateInfo*>(pCreate->lpCreateParams);
        SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pState);
    }
    else
    {
        pState = GetAppState(hwnd);
    }

    switch (uMsg)
    {


    // Remainder of the window procedure not shown ...

    }
    return TRUE;
}
```

An Object-Oriented Approach

We can extend this approach further. We have already defined a data structure to hold state information about the window. It makes sense to provide this data structure with member functions (methods) that operate on the data. This naturally leads to a design where the structure (or class) is responsible for all of the operations on the window. The window procedure would then become part of the class.

In other words, we would like to go from this:

C++

```
// pseudocode

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
    StateInfo *pState;

    /* Get pState from the HWND. */

    switch (uMsg)
    {
        case WM_SIZE:
            HandleResize(pState, ...);
            break;

        case WM_PAINT:
            HandlePaint(pState, ...);
            break;

       // And so forth.
    }
}
```

To this:

C++

```
// pseudocode

LRESULT MyWindow::WindowProc(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_SIZE:
            this->HandleResize(...);
            break;

        case WM_PAINT:
            this->HandlePaint(...);
            break;
    }
}
```

The only problem is how to hook up the `MyWindow::WindowProc` method. The RegisterClass function expects the window procedure to be a function pointer. You can't pass a pointer to a (non-static) member function in this context. However, you can pass a pointer to a *static* member function and then delegate to the member function. Here is a class template that shows this approach:

C++

```
template <class DERIVED_TYPE>
class BaseWindow
{
public:
    static LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
    {
        DERIVED_TYPE *pThis = NULL;
        if (uMsg == WM_NCCREATE)
        {
            CREATESTRUCT* pCreate = (CREATESTRUCT*)lParam;
            pThis = (DERIVED_TYPE*)pCreate->lpCreateParams;
            SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pThis);

            pThis->m_hwnd = hwnd;
        }
        else
        {
            pThis = (DERIVED_TYPE*)GetWindowLongPtr(hwnd, GWLP_USERDATA);
        }
        if (pThis)
        {
            return pThis->HandleMessage(uMsg, wParam, lParam);
        }
        else
        {
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
    }

    BaseWindow() : m_hwnd(NULL) { }

    BOOL Create(
        PCWSTR lpWindowName,
        DWORD dwStyle,
        DWORD dwExStyle = 0,
        int x = CW_USEDEFAULT,
        int y = CW_USEDEFAULT,
        int nWidth = CW_USEDEFAULT,
        int nHeight = CW_USEDEFAULT,
        HWND hWndParent = 0,
        HMENU hMenu = 0
        )
    {
        WNDCLASS wc = {0};

        wc.lpfnWndProc   = DERIVED_TYPE::WindowProc;
        wc.hInstance     = GetModuleHandle(NULL);
        wc.lpszClassName = ClassName();

        RegisterClass(&wc);

        m_hwnd = CreateWindowEx(
            dwExStyle, ClassName(), lpWindowName, dwStyle, x, y,
            nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this
            );

        return (m_hwnd ? TRUE : FALSE);
    }

    HWND Window() const { return m_hwnd; }

protected:
    virtual PCWSTR  ClassName() const = 0;
    virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam) =
0;

    HWND m_hwnd;
};
```

The `BaseWindow` class is an abstract base class, from which specific window classes are derived. For example, here is the declaration of a simple class derived from `BaseWindow`:

C++

```cpp
class MainWindow : public BaseWindow<MainWindow>
{
public:
    PCWSTR  ClassName() const { return L"Sample Window Class"; }
    LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);
};
```

To create the window, call `BaseWindow::Create`:

C++

```cpp
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int
nCmdShow)
{
    MainWindow win;

    if (!win.Create(L"Learn to Program Windows", WS_OVERLAPPEDWINDOW))
    {
        return 0;
    }

    ShowWindow(win.Window(), nCmdShow);

    // Run the message loop.

    MSG msg = { };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    return 0;
}
```

The pure-virtual `BaseWindow::HandleMessage` method is used to implement the window procedure. For example, the following implementation is equivalent to the window procedure shown at the start of Module 1.

C++

```
LRESULT MainWindow::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;

    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(m_hwnd, &ps);
            FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
            EndPaint(m_hwnd, &ps);
        }
        return 0;

    default:
        return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
    }
    return TRUE;
}
```

Notice that the window handle is stored in a member variable (*m_hwnd*), so we do not need to pass it as a parameter to `HandleMessage`.

Many of the existing Windows programming frameworks, such as Microsoft Foundation Classes (MFC) and Active Template Library (ATL), use approaches that are basically similar to the one shown here. Of course, a fully generalized framework such as MFC is more complex than this relatively simplistic example.

Next

Module 2: Using COM in Your Windows Program

Related topics

BaseWindow Sample

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A