Creating a Win32 Window Wrapper Class

by Oluseyi Sonaiya

Anyone familiar with the Win32 API knows that the process for creating windows is rather lengthy: you create a `WNDCLASSEX` structure, initialize its members and register it (per window class); create the window, check for the validity of the returned handle, and finally display the window. Of course, I neglected to mention that initializing the `WNDCLASSEX` structures members requires that you define a window procedure first...

So it's logical that the enterprising C++ developer would like to wrap all of the above in a nice class that would allow her write something like:

```
Window *wnd = new Window(...);
wnd->Create(...);
wnd->Show(...);
// use window
// window's destructor is automatically called, and
// performs necessary cleanup
```

Doesn't look to hard... **Ha!** It took me a good 2 hours to perfect the object association and proper message pump termination. I'll share my design principles, implementation, revisions and discoveries in the process of creating this wrapper class. I will not be covering the fundamentals of creating windows with Win32, or any other technology I consider to be elementary. There are several good references available on the web for these topics; please take advantage of them.

Without further ado, let's dive in!

Design Principles

In writing a Win32 window wrapper class, I wanted a method that would allow for as little coupling into the main application as possible, that would require the minimum user intervention necessary yet be extremely flexible. I also wanted the user to be able to extend the class without having to inherit or reimplement any methods. Most methods of wrapping windows in Win32 that I have seen on the web require the user to implement a full window procedure and handle all messages she is interested in (you may notice my habitual use of the singular feminine pronoun; deal with it). Microsoft's MFC and Borland's VCL use preprocessor "message maps", restricting the flexibility of the window class in terms of dynamic runtime modification since they are evaluated at compile time. Given the advent of the Standard Template Library and its integral part in the C++ language, I was sure I could find a container that would be as fast and flexible as the image in my mind.

As much as possible, I wanted defaults such that the wrappers could be used with minimal code. I also wanted the class to be sufficiently generic that it could be used for general-purpose applications as well as for high-performance applications like videogames. Over the course of the class development various features to accomplish this were added, and it was constantly interesting to find a way to integrate such features in a consistent way. Wherever possible and reasonable, I also tried to use names consistent with the Win32 API as a form of assitance to the developer.

The Window Class

Windows in Win32 are represented by *window handles*, so the logical first object to encapsulate is the window handle. Anybody who has tried to wrap window classes knows that you can not initialize the window class with a class method as the window procedure unless it is a static method, so the class also needs a static "message router" that determines which instance of the class is the recipient of the message and calls that instance's window procedure. So how exactly do we determine which instance should receive the message?

Every window has a 32-bit value associated with it intended for use by the application. This value can be set via the `SetWindowLong` function and retrieved with the `GetWindowLong`, and this is the core of the message routing technique.

```
LRESULT CALLBACK Window::MsgRouter(HWND hwnd, UINT message,
                                   WPARAM wparam, LPARAM lparam)
{
  Window *wnd = 0;

  // retrieve associated Window instance
  wnd = reinterpret_cast<Window *>(::GetWindowLong(hwnd, GWL_USERDATA));

  // call the windows message handler
  wnd->WndProc(message, wparam, lparam);
}
```

While this looks perfect, where and when was this value stored? At first I stored it during my `Window::Create` routine, but I ran into problems. You see, some messages are sent before `CreateWindowEx` returns - `WM_NCCREATE`, `WM_NCCALCSIZE` and `WM_CREATE` (in that order). If the value does not exist by the time `WM_NCCREATE` is called then, by some mysterious Windows behavior that I still don't understand, the value never gets inserted. The solution? The `WM_NCCREATE` message must be handled explicitly within the message router, with the object pointer being stored at this point instead. Okay, but where do we get the value from? The `CreateWindowEx` function allows the passing of a (void) pointer to window creation data. This data is made available to the window procedure as the `lparam` parameter in the form of an LPCREATESTRUCT. As an added precaution, we go ahead and store the window handle at this point.

```
LRESULT CALLBACK Window::MsgRouter(HWND hwnd, UINT message,
                                   WPARAM wparam, LPARAM lparam)
{
  Window *wnd = 0;

  if(message == WM_NCCREATE)
  {
    // retrieve Window instance from window creation data and associate
    wnd = reinterpret_cast<Window *>((LPCREATESTRUCT)lparam)->lpCreateParams;
    ::SetWindowLong(hwnd, GWL_USERDATA, reinterpret_cast<long>(wnd));

        // save window handle
    wnd->SetHWND(hwnd);
  }
  else
    // retrieve associated Window instance
    wnd = reinterpret_cast<Window *>(::GetWindowLong(hwnd, GWL_USERDATA));

  // call the windows message handler
  wnd->WndProc(message, wparam, lparam);
}
```

Message Handling

Alright! We've got our object properly associated with the window handle, and messages being properly routed. But what about that flexibility we mentioned earlier? As it stands, the window procedure has to handle all possible messages - an approach that would require reengineering the class for every new application. Not going to cut it.

In considering how to make this more flexible, I was struct by the fact that MFC and VCL call their methods of associating window messages with message handlers *message maps*. Being the STL afficionado that I am, that immediately struct a chord with me. *How's about I use a `std::map` to tie a particular message to a particular message handler?* That way I could insert *and replace* a message handler at any time without modifying the class.

For those of you not familar with the Standard Template Library (STL), I heartily recommend SGI's STL Documentation as both an introduction and a reference.

```
typedef long (* tyMessageHandler)(Window &, HWND, long, long);
typedef std::map<long, tyMessageHandler> tyMessageMap;
typedef tyMessageMap::iterator tyMessageIterator;
```

The Window class contains a single instance of `tyMessageMap`. This message map is then searched for the existence of a handler for a given message by the message router, and if none exist the default window procedure is invoked. I also chose to provide two (static) message handlers, partly as a template and partly to provide default functionality.

```
// Window::GetMessageHandler returns the address of the registered
// message handler if one exists
tyMessageIterator Window::GetMessageHandler(long message)
{
  // m_MsgHandlers is a tyMessageMap instance
  tyMessageIterator it = m_MsgHandlers.find(message);
  if(it = m_MsgHandlers.end())
    return NULL;
  return it;
}

// Window::OnClose is a static method called in response to WM_CLOSE
long Window::OnClose(Window &wnd, HWND hwnd, long param0, long param1)
{
  DestroyWindow(hwnd);
  return 0;
}
```

```
// Window::OnDestroy is a static method called in response to WM_DESTROY
long Window::OnDestroy(Window &wnd, HWND hwnd, long param0, long param1)
{
  PostQuitMessage(0);
  return 0;
}

// Final message handler version
LRESULT CALLBACK Window::MsgRouter(HWND hwnd, UINT message,
                                   WPARAM wparam, LPARAM lparam)
{
  Window *wnd = 0;

  if(message == WM_NCCREATE)
  {
    // retrieve Window instance from window creation data and associate
    wnd = reinterpret_cast<Window *>((LPCREATESTRUCT)lparam)->lpCreateParams;
    ::SetWindowLong(hwnd, GWL_USERDATA, reinterpret_cast<long>(wnd));

        // save window handle
    wnd->SetHWND(hwnd);
  }
  else
    // retrieve associated Window instance
    wnd = reinterpret_cast<Window *>(::GetWindowLong(hwnd, GWL_USERDATA));

  if(wnd)
  {
    tyMessageIterator it;
    it = wnd->GetMessageHandler(message);
    if(it != NULL)
      return (it->second)((*wnd), hwnd, wparam, lparam);
  }
  return DefWindowProc(hwnd, message, wparam, lparam);
}
```

Okay, so the message router takes care of object association, checks to see if there is an appropriate message handler and calls the default window procedure if there isn't. Fine. Now how do you add these message handlers? Behold the `Window::RegisterMessageHandler` method!

```
tyMessageHandler Window::RegisterMessageHandler(long message,
                                    tyMessageHandler handler)
{
  tyMessageHandler m = NULL;
  tyMessageIterator it = m_MsgHandlers.find(message);
  if(it != m_MsgHandlers.end())
    m = it->second;
  m_MsgHandlers.insert(std::pair<long,tyMessageHandler>(message, handler));
  return m;
}
```

Alright, so it wasn't so dramatic. The `RegisterMessageHandler` method inserts a message handler into the message map and returns the previous message handler, if there was one. That about wraps it up for message handling (I say about because I'll revisit one of the methods described above later). Now let's turn to integrating the Window with the application message pump.

The Window Class and the Application Message Pump

A typical Windows message pump looks something like this:

```
MSG msg;
while(GetMessage(&msg, hwnd, 0, 0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);

  // other procedures
}
```

Microsoft actually advises against this form of message loop because GetMessage has a tri-state return value and the above may cause an attempt at execution even when there was an error.

We do it anyway.

This article is getting long and I'm getting tired, so I'll dump the code and then break it down.

```
// Window::OnDestroy, revisited
long Window::OnDestroy(Window &wnd, HWND hwnd, long param0, long param1)
{
  PostQuitMessage(0);
  wnd->SetExit(true);
  return 0;
}

// Window::HandleMessage ties everything together
bool Window::HandleMessages()
{
  static MSG msg;

  if(!m_hwnd)
    throw std::runtime_error(std::string("Window not yet created"));

  if((m_UsePeekMessage)
     ? ::PeekMessage(&msg, m_hwnd, 0, 0, PM_REMOVE)
     : ::GetMessage(&msg, m_hwnd, 0, 0))
  {
    ::TranslateMessage(&msg);
    ::DispatchMessage(&msg);
  }
  else
  {
    if(IsExit())
    {
      SetExitCode((long)msg.lParam);
      return false;
    }
    if(m_UseWaitMessage)
    WaitMessage();
  }

  return true;
}
```

There's a bunch of functions not introduced here. Obviously, the constructor and methods like `Create` and `ShowWindow`; I leave this as an exercise for the inexperienced reader and a chore for the expert. There are also the boolean variables `m_UseWaitMessage`, `m_UsePeekMessage` and `m_Exit`; the exit code for the Window (to pass to the application if the Window is the main window); and the static method to register the windowclass. The code above is fairly self-explanatory. As is evident, I use exceptions to avoid having to pass and compare return values for application-terminating errors. I also use the `m_UsePeekMessage` and `m_UseWaitMessage` as discriminants between using `GetMessage` and `PeekMessage`, and whether or not to use `WaitMessage` respectively.

That's it. And here's a simple example of my implementation in use:

```
Window *g_wnd;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE /* hPrevInstance */,
                   LPSTR lpCmdLine, int nShowCmd)
{
  try
  {
    g_wnd = new Window(hInstance);
    g_wnd->Create(NULL, WS_OVERLAPPEDWINDOW|WS_VISIBLE);

    if(!g_wnd)
      throw std::runtime_error(std::string("Initialization Failed: Window::Create"));

    g_wnd->UsePeekMessage();
    g_wnd->UseWaitMessage(false);
    g_wnd->ShowWindow(nShowCmd);
    while(true)
    {
      if(!g_wnd->HandleMessages())
        break;
    }
  }
  catch(std::runtime_error &e)
  {
    ::MessageBox(NULL, e.what(), "Runtime Error",
                 MB_OK|MB_ICONERROR|MB_DEFBUTTON1|MB_TASKMODAL|MB_TOPMOST);
    return -1;
  }
  catch(std::logic_error &e)
  {
    ::MessageBox(NULL, e.what(), "Logic Error",
                 MB_OK|MB_ICONERROR|MB_DEFBUTTON1|MB_TASKMODAL|MB_TOPMOST);
    return -1;
  }
  catch(...)
  {
    ::MessageBox(NULL, "Unhandled Exception", "Unknown Error",
                 MB_OK|MB_ICONERROR|MB_DEFBUTTON1|MB_TASKMODAL|MB_TOPMOST);
    return -1;
  }

  return g_wnd->ExitCode();
}
```

Hopefully this article has been useful as a solution, but more importantly has inspired you to even better implementations which I hope you share with us all.

Cheers!

Discuss this article in the forums

Date this article was posted to GameDev.net: **4/29/2002**
(Note that this date does not necessarily correspond to the date the article was written)

See Also:
Featured Articles
Windows