When the normal window destruction messages are thrown for a loop

devblogs.microsoft.com/oldnewthing/20050727-16

July 27, 2005

Raymond Chen

Last time, I alluded to weirdness that can result in the normal cycle of destruction messages being thrown out of kilter.

Commenter Adrian noted that the `WM_GETMINMAXINFO` message arrives before `WM_NCCREATE` for top-level windows. This is indeed unfortunate but (mistake or not) it's been that way for over a decade and changing it now would introduce serious compatibility risk.

But that's not the weirdness I had in mind.

Some time ago I was helping to debug a problem with a program that was using the ListView control, and the problem was traced to the program subclassing the ListView control and, through a complicated chain of C++ objects, ending up attempting to destroy the ListView control while it was already in the process of being destroyed.

Let's take our new scratch program and illustrate what happens in a more obvious manner.

```
class RootWindow : public Window
{
public:
 RootWindow() : m_cRecurse(0) { }
 …
private:
 void CheckWindow(LPCTSTR pszMessage) {
  OutputDebugString(pszMessage);
  if (IsWindow(m_hwnd)) {
   OutputDebugString(TEXT(” – window still exists\r\n”));
  } else {
   OutputDebugString(TEXT(” – window no longer exists\r\n”));
  }
 }
private:
 HWND m_hwndChild;
 UINT m_cRecurse;
 …
};


LRESULT RootWindow::HandleMessage(
                          UINT uMsg, WPARAM wParam, LPARAM lParam)
{
 …
  case WM_NCDESTROY:
   CheckWindow(TEXT(“WM_NCDESTROY received”));
   if (m_cRecurse < 2) {
    m_cRecurse++;
    CheckWindow(TEXT(“WM_NCDESTROY recursing”));
    DestroyWindow(m_hwnd);
    CheckWindow(TEXT(“WM_NCDESTROY recursion returned”));
   }
   PostQuitMessage(0);
   break;


  case WM_DESTROY:
   CheckWindow(TEXT(“WM_DESTROY received”));
   if (m_cRecurse < 1) {
    m_cRecurse++;
    CheckWindow(TEXT(“WM_DESTROY recursing”));
    DestroyWindow(m_hwnd);
    CheckWindow(TEXT(“WM_DESTROY recursion returned”));
   }
   break;
  …
}
```

We add some debug traces to make it easier to see what is going on. Run the program, then close it, and watch what happens.

```
WM_DESTROY received - window still exists
WM_DESTROY recursing - window still exists
WM_DESTROY received - window still exists
WM_NCDESTROY received - window still exists
WM_NCDESTROY recursing - window still exists
WM_DESTROY received - window still exists
WM_NCDESTROY received - window still exists
WM_NCDESTROY recursion returned - window no longer exists
Access violation - code c0000005
eax=00267160 ebx=00000000 ecx=00263f40 edx=7c90eb94 esi=00263f40 edi=00000000
eip=0003008f esp=0006f72c ebp=0006f73c iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000283
0003008f ??              ???
```

Yikes! What happened?

When you clicked the "X" button, this started the window destruction process. As is to be expected, the window received a `WM_DESTROY` message, but the program responds to this by attempting to destroy the window **again**. Notice that `IsWindow` reported that the window still exists at this point. This is true: The window does still exist, although it happens to be in the process of being destroyed. In the original scenario, the code that destroyed the window went something like

```
if (IsWindow(hwndToDestroy)) {
 DestroyWindow(hwndToDestroy);
}
```

At any rate, the recursive call to `DestroyWindow` caused a **new** window destruction cycle to begin, nested inside the first one. This generates a new `WM_DESTROY` message, followed by a `WM_NCDESTROY` message. (Notice that this window has now received **two** `WM_DESTROY` messages!) Our bizarro code then makes yet another recursive call to `DestroyWindow`, which starts a **third** window destruction cycle. The window gets its third `WM_DESTROY` message, then its second `WM_NCDESTROY` message, at which point the second recursive call to `DestroyWindow` returns. At this point, the window no longer exists: `DestroyWindow` has destroyed the window.

And that's why we crash. The base `Window` class handles the `WM_NCDESTROY` message by destroying the instance variables associated with the window. Therefore, when the innermost `DestroyWindow` returns, the instance variables have been thrown away. Execution then resumes with the base class's `WM_NCDESTROY` handler, which tries to access the instance variables and gets heap garbage, and then makes the even worse no-no of freeing memory that is already freed, thereby corrupting the heap. It is here that we crash, attempting to call the virtual destructor on an already-destructed object.

I intentionally chose to use the new scratch program (which uses C++ objects) instead of the classic scratch program (which uses global variables) to highlight the fact that after the recursive `DestroyWindow` call, all the instance variables are gone and you are operating on

freed memory.

Moral of the story: Understand your window lifetimes and don't destroy a window that you know already to be in the process of destruction.

Raymond Chen

Follow