©2009 Microsoft Corporation. All rights reserved.

C++ Q&A

System Tray Balloon Tips and Freeing Resources Quickly in .NET

Paul DiLascia

Code download available at: CQA0211.exe (62 KB)
Browse the Code Online [http://msdn.microsoft.com/en-us/magazine/cc164954.aspx]

Q I have a pretty simple, but still irritating problem. I don't know how to make a tray icon show a tooltip without moving the mouse over the icon. Is there some kind of message I can send to the tray to make the system show the tooltip?
Q I have a pretty simple, but still irritating problem. I don't know how to make a tray icon show a tooltip without moving the mouse over the icon. Is there some kind of message I can send to the tray to make the system show the tooltip?
Deyan Dyankov

A There are two kinds of tips associated with tray icons: an ordinary old-style tooltip that appears when the user moves the mouse over your tray icon and the newfangled balloon tip that appears when directed by your program. Balloon tips resemble the word balloons that appear in comic strips. **Figure 1** shows an example; here, Windows® is telling me it stopped my USB Memory Card Reader. Balloon tips provide a non-intrusive way for tray programs to notify users when something happens. But how do you make a balloon tip appear? All tray icon behavior operates through a single API function, Shell_NotifyIcon. This function takes a struct called NOTIFYICONDATA that contains fields to tell Windows what you want to do. If you're interested, I wrote about tray icons in *Microsoft® Systems Journal* way back in March 1996 and again in February 1999 (see the March 1996 and February 1999 columns).
A There are two kinds of tips associated with tray icons: an ordinary old-style tooltip that appears when the user moves the mouse over your tray icon and the newfangled balloon tip that appears when directed by your program. Balloon tips resemble the word balloons that appear in comic strips. **Figure 1** shows an example; here, Windows® is telling me it stopped my USB Memory Card Reader. Balloon tips provide a non-intrusive way for tray programs to notify users when something happens. But how do you make a balloon tip appear? All tray icon behavior operates through a single API function, Shell_NotifyIcon. This function takes a struct called NOTIFYICONDATA that contains fields to tell Windows what you want to do. If you're interested, I wrote about tray icons in *Microsoft® Systems Journal* way back in March 1996 and again in February 1999 (see the March 1996 and February 1999 columns).

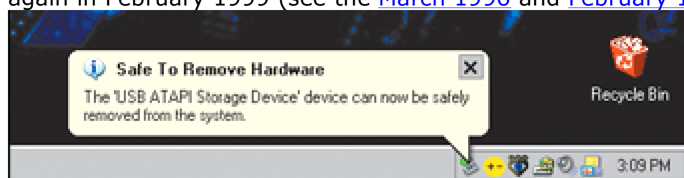


Figure 1 **Balloon Tip Example**

One of the flags in NOTIFYICONDATA.uFlags is NIF_TIP, used to set the ordinary, mouseover tooltip. A new flag (available where _WIN32_IE >= 0x0500), NIF_INFO, lets you display a balloon tip. In other words, to display a balloon tip, you must call Shell_NotifyIcon with NIF_INFO. The text goes in szInfo, and you can put a title in szInfoTitle. You can even specify a timeout value in NOTIFYICONDATA.uTimeout: Windows dispatches your balloon automatically after the specified number of milliseconds elapses.
To show how it all works in practice, I modified the CTrayIcon and the TrayTest sample from my February 1999 *MSJ* column.
To implement the balloons, CTrayIcon gets a new method called ShowBalloonTip that comes in two overloaded flavors: you can give it either a text string or resource ID. The resource ID option loads the string and calls the text version, which looks like this:

Copy Code

```
BOOL CTrayIcon::ShowBalloonTip(LPCTSTR szMsg,
  LPCTSTR szTitle, UINT uTimeout, DWORD dwInfoFlags)
{
  m_nid.cbSize=sizeof(NOTIFYICONDATA);
  m_nid.uFlags = NIF_INFO;
  m_nid.uTimeout = uTimeout;
  m_nid.dwInfoFlags = dwInfoFlags;
  strcpy(m_nid.szInfo,szMsg ? szMsg : _T(""));
  strcpy(m_nid.szInfoTitle,szTitle ? szTitle : _T(""));
  return Shell_NotifyIcon(NIM_MODIFY, &m_nid);
}
```

It's all very straightforward and boring: copy the text into the buffer and yadda yadda... By default, dwInfoFlags is set to NIIF_INFO to display an information icon next to the text; other possibilities are NIIF_ERROR for an error, NIIF_WARNING for a warning, and NIIF_NONE for no icon at all. **Figure 2** shows TrayTest in action with the balloon tip visible; **Figure 3** shows only the relevant changes to TrayTest since the original 1999 column. As always, you can download full source from the link at the top of this article.

Figure 3 TrayTest Update

TrayIcon.h

Copy Code

```
#pragma once
#include "Subclass.h"

// CTrayIcon manages an icon in the Windows system tray.
// The sample program TRAYTEST shows how to use it.
class CTrayIcon : public CCmdTarget {
public:

   // Show balloon tip
   BOOL ShowBalloonTip(LPCTSTR szMsg, LPCTSTR szTitle,
      UINT uTimeout, DWORD dwInfoFlags=NIIF_INFO);

   // Show balloon tip: use resource ID instead of LPCSTR.
   BOOL ShowBalloonTip(UINT uID, LPCTSTR szTitle,
      UINT uTimeout, DWORD dwInfoFlags=NIIF_INFO);

};
```

TrayIcon.cpp

Copy Code

```
#include "stdafx.h"
#include "trayicon.h"
#include <afxpriv.h>    // for AfxLoadString

#define countof(x)   (sizeof(x)/sizeof(x[0]))

// Windows sends this message when the taskbar is created. This can
// happen if it crashes and Windows has to restart it. CTrayIcon
// responds by reinstalling its icon.
const UINT WM_TASKBARCREATED = ::RegisterWindowMessage(_T("TaskbarCreated"));

IMPLEMENT_DYNAMIC(CTrayIcon, CCmdTarget)

CTrayIcon::CTrayIcon(UINT uID){
   // Initialize NOTIFYICONDATA
   memset(&m_nid, 0 , sizeof(m_nid));
   m_nid.cbSize = sizeof(m_nid);
   m_nid.uID = uID;  // never changes after construction

   m_notifyHook.m_pTrayIcon = this; // notification window hook
   m_parentHook.m_pTrayIcon = this; // parent window hook

   // Use resource string as tip if there is one
   AfxLoadString(uID, m_nid.szTip, sizeof(m_nid.szTip));
}

CTrayIcon::~CTrayIcon(){
   SetIcon(0); // remove icon from system tray
}

// Set notification window. It must be created already.
void CTrayIcon::SetNotificationWnd(CWnd* pNotifyWnd, UINT uCbMsg){
   // If the following assert fails, you're probably
   // calling me before you created your window. Oops.
   ASSERT(pNotifyWnd==NULL || ::IsWindow(pNotifyWnd->GetSafeHwnd()));
   m_nid.hWnd = pNotifyWnd->GetSafeHwnd();

   ASSERT(uCbMsg==0 || uCbMsg>=WM_USER);
   m_nid.uCallbackMessage = uCbMsg;

   CWnd* pParentWnd = pNotifyWnd ? pNotifyWnd->GetTopLevelParent() :
     NULL;

   // Install window hooks. Must be different because
   // taskbar creation message only goes to top-level parent.
   m_notifyHook.HookWindow(pNotifyWnd);
   if (pParentWnd!=pNotifyWnd)
      m_parentHook.HookWindow(pParentWnd);
}

// This is the main variant for setting the icon. Sets both the icon and
// tooltip from resource ID. To remove the icon, call SetIcon(0)
BOOL CTrayIcon::SetIcon(UINT uID){
   HICON hicon=NULL;
```

```
   if (uID) {
      AfxLoadString(uID, m_nid.szTip, sizeof(m_nid.szTip));
      hicon = AfxGetApp()->LoadIcon(uID);
   }
   return SetIcon(hicon, NULL);
}

// Common SetIcon for all overloads.
BOOL CTrayIcon::SetIcon(HICON hicon, LPCTSTR lpTip){
   UINT msg;
   m_nid.uFlags = 0;

   // Set the icon
   if (hicon) {
      // Add or replace icon in system tray
      msg = m_nid.hIcon ? NIM_MODIFY : NIM_ADD;
      m_nid.hIcon = hicon;
      m_nid.uFlags |= NIF_ICON;
   } else { // remove icon from tray
      if (m_nid.hIcon==NULL)
         return TRUE;      // already deleted
      msg = NIM_DELETE;
   }

   // Use the tip, if any
   if (lpTip)
      _tcsncpy(m_nid.szTip, lpTip, countof(m_nid.szTip));
   if (m_nid.szTip[0])
      m_nid.uFlags |= NIF_TIP;

   // Use callback if any
   if (m_nid.uCallbackMessage && m_nid.hWnd)
      m_nid.uFlags |= NIF_MESSAGE;

   BOOL bRet = Shell_NotifyIcon(msg, &m_nid);
   if (msg==NIM_DELETE || !bRet)
      m_nid.hIcon = NULL;  // failed
   return bRet;
}

// Show balloon tip: args give message, timeout, etc.
BOOL CTrayIcon::ShowBalloonTip(LPCTSTR szMsg, LPCTSTR szTitle,
   UINT uTimeout, DWORD dwInfoFlags){
   m_nid.cbSize=sizeof(NOTIFYICONDATA);
   m_nid.uFlags = NIF_INFO;
   m_nid.uTimeout = uTimeout;
   m_nid.dwInfoFlags = dwInfoFlags;
   strcpy(m_nid.szInfo,szMsg ? szMsg : _T(""));
   strcpy(m_nid.szInfoTitle,szTitle ? szTitle : _T(""));
   return Shell_NotifyIcon(NIM_MODIFY, &m_nid);
}

BOOL CTrayIcon::ShowBalloonTip(UINT uID, LPCTSTR szTitle,
   UINT uTimeout, DWORD dwInfoFlags){
   CString s;
   return s.LoadString(uID) ?
      ShowBalloonTip(s, szTitle, uTimeout, dwInfoFlags) : FALSE;
}

// Same hook class used for both notification window and top-level
// parent; hook function determines which.
LRESULT CTrayIcon::CTrayHook::WindowProc(UINT msg, WPARAM wp, LPARAM lp)
{
   if (msg==m_pTrayIcon->m_nid.uCallbackMessage &&
       m_hWnd==m_pTrayIcon->m_nid.hWnd) {

      m_pTrayIcon->OnTrayNotify(wp, lp);

// } else if (msg==WM_DESTROY) {
//    m_pTrayIcon->SetIcon(NULL);

   } else if (msg==WM_TASKBARCREATED) {
      m_pTrayIcon->OnTaskBarCreate(wp, lp);
   }
   return CSubclassWnd::WindowProc(msg, wp, lp);
}
```

```
// Default event handler handles right-menu and double-click.
// Override to do something different.
LRESULT CTrayIcon::OnTrayNotify(WPARAM wID, LPARAM lEvent){
   if (wID!=m_nid.uID ||
      (lEvent!=WM_RBUTTONUP && lEvent!=WM_LBUTTONDBLCLK))
      return 0;

   // If there's a resource menu with the same ID as the icon, use it as
   // the right-button popup menu. CTrayIcon will interpret the first
   // item in the menu as the default command for WM_LBUTTONDBLCLK
   CMenu menu;
   if (!menu.LoadMenu(m_nid.uID))
      return 0;
   CMenu* pSubMenu = menu.GetSubMenu(0);
   if (!pSubMenu)
      return 0;

   if (lEvent==WM_RBUTTONUP) {

      // Make first menu item the default (bold font)
      ::SetMenuDefaultItem(pSubMenu->m_hMenu, 0, TRUE);

      // Display the menu at the current mouse location. A "bug" in
      // Windows 95 that requires calling SetForegroundWindow. To find
      // out more, read Q135788 in MSDN.
      CPoint mouse;
      GetCursorPos(&mouse);
      ::SetForegroundWindow(m_nid.hWnd);
      ::TrackPopupMenu(pSubMenu->m_hMenu, 0, mouse.x, mouse.y, 0,
         m_nid.hWnd, NULL);
      ::PostMessage(m_nid.hWnd, WM_NULL, 0, 0);

   } else  // double-click: execute first menu item
    ::SendMessage(m_nid.hWnd, WM_COMMAND, pSubMenu->GetMenuItemID(0), 0);

   return 1; // handled
}

// Explorer had to restart the taskbar: add icons again
LRESULT CTrayIcon::OnTaskBarCreate(WPARAM wp, LPARAM lp){
   // Reinstall taskbar icon
   HICON hIcon = m_nid.hIcon;
   m_nid.hIcon = NULL;
   if (hIcon)
      SetIcon(hIcon, NULL); // will reuse current tip
   return 0;
}
```

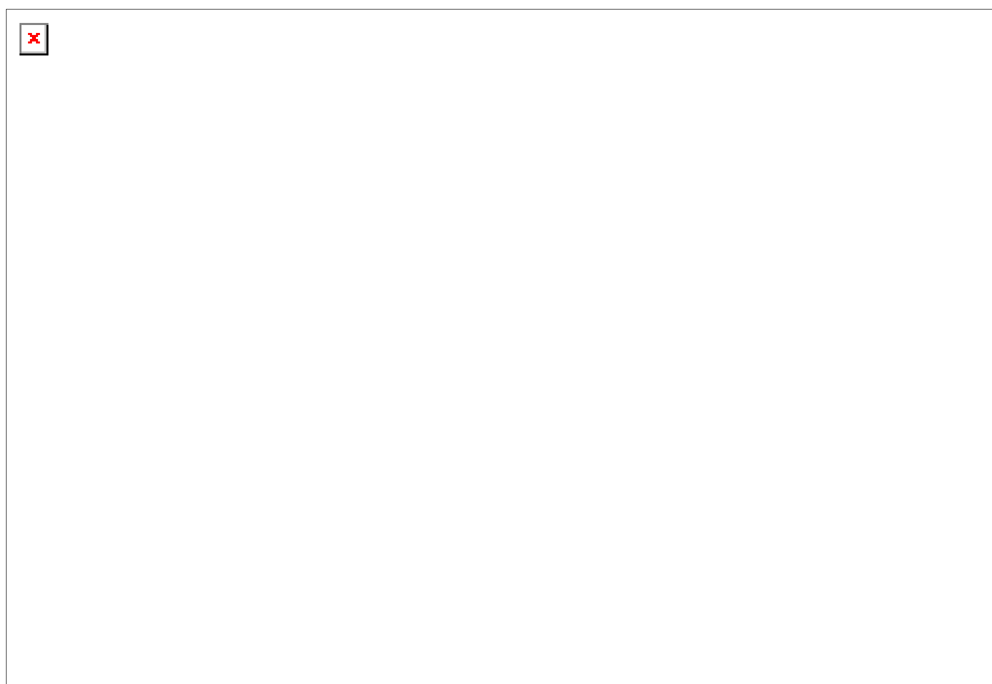
MainFrm.cpp

Copy Code

```
#include "stdafx.h"
#include "TrayTest.h"
#include "mainfrm.h"

// Message ID used for tray notifications
#define WM_MY_TRAY_NOTIFICATION WM_USER+0

IMPLEMENT_DYNAMIC(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
   ON_MESSAGE(WM_MY_TRAY_NOTIFICATION, OnTrayNotification)
   ON_COMMAND(ID_VIEW_SHOWBALLOONTIP, OnShowBalloonTip)
END_MESSAGE_MAP()

•••// standard MFC stuff

#ifdef  NIN_BALLOONSHOW
#define NIN_BALLOONFIRST NIN_BALLOONSHOW
#define NIN_BALLOONLAST  NIN_BALLOONUSERCLICK
#endif

// Handle notification from tray icon: display a message.
LRESULT CMainFrame::OnTrayNotification(WPARAM uID, LPARAM lEvent){
   if (m_bShowTrayNotifications) {
      CString msg;
```

```
      if (WM_MOUSEFIRST<=lEvent && lEvent<=WM_MOUSELAST) {
         static LPCSTR MouseMsgs[] = { _T("WM_MOUSEMOVE"),
            _T("WM_LBUTTONDOWN"), _T("WM_LBUTTONUP"),
            _T("WM_LBUTTONDBLCLK"), _T("WM_RBUTTONDOWN"),
            _T("WM_RBUTTONUP"), _T("WM_RBUTTONDBLCLK"),
            _T("WM_MBUTTONDOWN"), _T("WM_MBUTTONUP"),
            _T("WM_MBUTTONDBLCLK") };
         msg = MouseMsgs[lEvent-WM_MOUSEFIRST];

#ifdef NIN_BALLOONSHOW
      } else if (NIN_BALLOONFIRST<=lEvent && lEvent<=NIN_BALLOONLAST) {
         static LPCSTR BalloonMsgs[] = { _T("NIN_BALLOONSHOW"),
            _T("NIN_BALLOONHIDE"), _T("NIN_BALLOONTIMEOUT"),
            _T("NIN_BALLOONUSERCLICK") };
         msg = BalloonMsgs[lEvent-NIN_BALLOONFIRST];
#endif
      } else
         msg = _T("(Unknown)");

      CString s;
      s.Format(_T("Tray notification: ID=%d, lEvent=0x%04x %s\r\n"),
         uID, lEvent, (LPCTSTR)msg);

      m_wndEdit.SetSel(-1, -1);     // end of edit text
      m_wndEdit.ReplaceSel(s);      // append string..
      m_wndEdit.SendMessage(EM_SCROLLCARET); // ..and make visible
   }
   return 0;
}

// Show balloon tip: call CTrayIcon to do it
void CMainFrame::OnShowBalloonTip(){
   m_trayIcon.ShowBalloonTip(IDS_BALLOONTIP, "TrayTest", 4000);
}
```

Figure 2 TrayTest with Balloon Tip



There's only one way to display the balloon tip (Shell_NotifyIcon), but there are several ways the tip can terminate. The user can click, either on the balloon itself or its close box, or Windows can terminate the tip when the timeout expires. How do you know which event happened? When you create your tray icon, you can supply an HWND and message ID to receive notifications when something happens. If the user clicks the balloon tip, Windows sends NIN_BALLOONUSERCLICK; if the timeout expires or the user clicks close, Windows sends NIN_BALLOONTIMEOUT. As far as I know, there's no way to distinguish between an actual timeout and the user clicking the close icon. **Figure 4** shows all the balloon-related tray notifications.

Figure 4 Tray Notifications

Notification	Description
NIN_BALLOONSHOW	Sent when the balloon is shown.
NIN_BALLOONHIDE	Sent when the balloon disappears; for example, when the icon is deleted. This message

	is not sent if the balloon is dismissed because of a timeout or mouse click by the user.
NIN_BALLOONTIMEOUT	Sent when the balloon is dismissed because of a timeout, or because the user clicked on the balloon close box (X).
NIN_BALLOONUSERCLICK	Sent when the user clicks the mouse on the balloon tip or tray icon (while balloon is displayed).

You can use TrayTest to see the notifications. In fact, that's what TrayTest does: display tray notifications as they arrive. To see the notifications, run TrayTest, click OK to the initial dialog, then double-click the banana tray icon to invoke the window in **Figure 1**. Use View | Show Balloon Tip to make TrayTest display its tip, then watch what messages appear in the main window as you close the tip or wait for the timeout. One peculiar phenomenon I discovered while exploring: the balloon won't timeout as long as your tray app has focus. Apparently, the meter doesn't start until you switch to another app.
A final balloon caution: please use balloon tips and tray icons sparingly! Don't implement a tray icon, as so many programmers do, just to be cute. Only do it if there's a genuine need. Users will find your program annoying and will most likely uninstall it if you bombard them with balloons and other screen junk. If you really must implement a tray icon, at least give users the option of turning it off.

QI'm a little confused about C# nondeterministic destruction and its implications. I've read that C# destructors are not always called. Is this true? When am I supposed to implement Dispose?
QI'm a little confused about C# nondeterministic destruction and its implications. I've read that C# destructors are not always called. Is this true? When am I supposed to implement Dispose?
Several readers

AFor those readers who've had their heads in the sand for the past two years or who have deliberately avoided anything with .NET in the title, one of the most controversial features of C# is what the OOPpundits call nondeterministic destruction. I hate to contribute to the already ponderous accumulation of material on this subject, but since it's so fundamental to the Microsoft .NET Framework, a quick review seems in order.
AFor those readers who've had their heads in the sand for the past two years or who have deliberately avoided anything with .NET in the title, one of the most controversial features of C# is what the OOPpundits call nondeterministic destruction. I hate to contribute to the already ponderous accumulation of material on this subject, but since it's so fundamental to the Microsoft .NET Framework, a quick review seems in order.
So what is nondeterministic destruction? It's not a religious scenario for Armageddon; it's a particular policy of destroying objects in memory. In C/C++, you call new to allocate an object and delete to free it. The compiler generates code to call your constructor when you allocate, and to call your destructor when you free an object; likewise for stack objects when they go in and out of scope. In official jabberwocky, C++ programmers say, "C++ destruction is deterministic" because your destructor is guaranteed to be called immediately when you delete your object or it goes out of scope, before the next thread code line executes.
In C# you allocate with new, but you never free objects explicitly because that's the garbage collector's job. .NET keeps meticulous track of who's using what, and when an object becomes inactive because nobody's using it (nothing points to it) that object becomes a candidate for recycling. But .NET doesn't actually recycle the object until the garbage collector decides to collect, which could be a while. .NET programmers say "destruction is nondeterministic" because you can't determine when your destructor/Finalize method will be called, only that it will be called.
The .NET approach works great for memory, where free pools and allocation can be optimized for the ways most programs use memory; for example, using lots of little objects is very fast. But problems arise when you have some other kind of resource like a file handle or window. You'd like to free the resource as soon as you're done, but .NET doesn't call your destructor (Finalize method) right away. In fact, it takes two garbage runs to fully free an object with Finalize: one pass is required to call Finalize, and another pass to free the memory.
Many C++ programmers find nondeterministic destruction downright primitive. They've long grown accustomed to encapsulating initialization/termination sequences in their constructors/destructors. For example, consider a simple but typical example: CWaitCursor in MFC. This class has a constructor that sets the cursor to an hourglass and a destructor that restores it to its original shape. Anytime you want to show an hourglass while your app goes temporarily computational, all you have to do is instantiate one of these doohickeys:

Copy Code

```
void CalcPiTo1000Digits()
{
  CWaitCursor wc; // display hourglass
  •••
}
```

One line—short and sweet. The ctor/dtor pattern is the basis for smart pointers and many other useful C++ classes, so ubiquitous and helpful that grown C++ programmers have been known to weep over the loss of deterministic destruction in .NET. To help alleviate their woe, the friendly Redmondtonians prescribe something called the Dispose pattern. Objects that require disposal implement IDisposable, an interface with just one method:

Copy Code

```
interface IDisposable
{
  public void Dispose();
}
```

If your class uses resources that must be released immediately in order to conserve them, you should free them in Dispose. The calling code is responsible for calling Dispose at the right time. The full Dispose pattern is actually more complex than it might at first seem. For example, there's an overloaded Dispose(bool) method with a Boolean argument

that's true when called from Dispose but false when called from Finalize (destructor); you must write Dispose in such a way that it can be called repeatedly without crashing or throwing an exception; you can't call other objects when the disposing flag is false; Dispose should always call Base.Dispose; and class methods should fail gracefully when called after the object is disposed. This means that technically you should preface every method with the following code:

Copy Code

```
if (disposed)
  throw ObjectDisposedException;
```

Yikes! In practice, few programmers bother. Oh, and did I mention you should call GC.SuppressFinalize so the garbage collector doesn't needlessly Finalize your object? For the full Dispose spec, see "Implementing Finalize and Dispose to Clean Up Unmanaged Resources" and "Programming for Garbage Collection" in the .NET documentation.
Personally, I find the Dispose pattern a bit amusing because the whole reason for having garbage collection is so programmers don't have to remember to call delete. But now we must remember to implement Dispose with all its elaborate rules and call it at the appropriate time. Well, hey—whatever. It just goes to show that no matter how hard people try to build a system that's bug proof, programming will always require common sense. If you can't remember to clean up after yourself, should you be programming?
The Dispose pattern is a programming convention, not a language feature. It's up to you to implement IDisposable and call Dispose when you want to free resources. However, there is one place where Dispose is actually built into C#:

Copy Code

```
using (MyClass d = new MyClass()) {
  d.DoSomething();
} // compiler calls d.Dispose() here if
  // MyClass implements IDisposable.
```

If MyClass implements IDisposable, the compiler inserts a call to d.Dispose at the end of the code block and thus provides nearly the same behavior as C++, at least as far as destruction goes. You can list multiple variables inside using () if you separate them with commas.
Figure 5 shows a little class I wrote, Hourglass, that implements something like CWaitCursor for C#. To use it, write:

Copy Code

```
void CalcPiTo1000Digits()
{
  using (new Hourglass()) {
    // calculate...
  }
}
```

It's not quite as elegant as C++, but close. You needn't assign the Hourglass to a variable; just creating it is enough to trigger Dispose at the end of the using block. Alas, using is available in C#, not Visual Basic. Nyah, nyah.

Figure 5 Hourglass

Copy Code

```
using System;
using System.Windows.Forms;

// Handy hourglass class to display an hourglass during long operation
// Syntax:
// using (new Hourglass()) {
//     // long operation here
// }
// Implements IDisposable to work in conjunction with C# 'using'
// keyword.

public class Hourglass : IDisposable
{
   private Cursor oldCursor;

   public Hourglass() : this(Cursors.WaitCursor)
   {

   }

   public Hourglass(Cursor cursor)
   {
      oldCursor = Cursor.Current;
      Cursor.Current = cursor;
   }

   public void Dispose()
   {
      if (oldCursor!=null)
```

```
        {
           Cursor.Current = oldCursor;
           oldCursor = null;
        }
     }
  }
```

If you want to read more about why .NET destruction works the way it does, the following link provides an excellent summary: Resource Management in .NET. Or, instead, you can search Google for the words "Brian Harry Resource Management." Happy programming!

Send questions and comments for Paul to cppqa@microsoft.com.

Paul DiLasciais a freelance writer, consultant, and Web/UI designer-at-large. He is the author of *Windows++: Writing Reusable Windows Code in C++* (Addison-Wesley, 1992). Paul can be reached at askpd@pobox.com or http://www.dilascia.com.