# Keep a Window Visible

Prevent users from moving windows offscreen, work with unsafe and unchecked code, and make a field variable declaration behave like a constant.

by Karl E. Peterson, Juval Löwy, and Mattias Sjögren

**Technology Toolbox**

- ☑ VB.NET
- ☑ C#
- ☐ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☑ VB6
- ☑ **Note:** Karl E. Peterson's solution also works with VB5.

## Q: Keep Your Window Visible

How can I prevent a user from moving a window off the screen? I'd like to allow users to move a window around the desktop at will, but not allow them to move any part of the window off the desktop. The entire window must be viewable at all times.

## A:

Hopefully, this is a client specification and not a method to keep some sort of popup advertisement perpetually in someone's face. (Online, a <g> would probably follow, but irritating users is rarely funny.) Assuming a legitimate need, the simple answer is that you'd want to hook your form's message stream and respond to incoming WM_MOVING messages (see Additional Resources). If you don't have a favorite drop-in subclassing module, I'd urge you to grab HookMe.zip from my Web site or this column's sample code (download the code from the *VSM* Web site; see the Go Online box for details).

Windows sends WM_MOVING messages to a window immediately prior to the user getting any feedback. These messages are accompanied by a pointer to a RECT structure in lParam that contains the drag rectangle coordinates Windows displays to the user. You're only given a pointer, so you need to copy the data at this address to a RECT structure declared within your hook procedure (see Listing 1).

At this point, you're free to examine the RECT coordinates and even modify them to suit your needs. In this case, you'd want to ensure that none of the edges go past the edge of the screen, and if they do, correct them to remain onscreen. After any necessary modifications, copy the updated structure back to the same address passed in lParam and tell Windows you've handled the message by returning True for the function result. —*K.E.P.*

## Q: Work With Unsafe Code

I hear you can do pointers in C#, and that you can manipulate memory that way. How is it done? Is the memory still managed by .NET? Is the code still managed code?

## A:

C# does support direct memory manipulation using pointers. Such C# code is called unsafe code, because this code lets go of most of the safety of .NET memory management. However, unsafe code is still managed code, because it runs in the Common Language Runtime (CLR), and .NET still garbage collects it. C# supports unsafe code to ease the task of porting legacy C++ applications to C# in cases that use

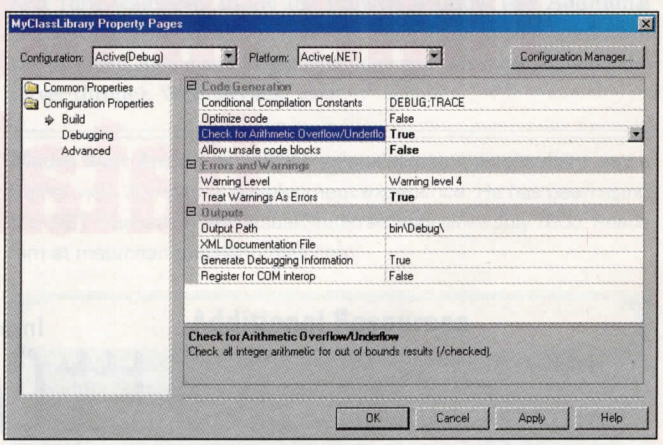


**Figure 1 Configure Project Settings.** The Properties page lets you configure project settings, including warning level, unsafe code support, unchecked code support, and treating warnings as errors. You can configure different settings for Debug and Release builds.

## Go Online!

Use these Locator+ codes at **www.visualstudiomagazine.com** to go directly to these resources.

**Download**

**VS0209QA** Download the code for this article, which includes VB6 code demonstrating how to keep a form fully onscreen, and VB.NET and C# code showing how to use read-only variables.

**Discuss**

**VS0209QA_D** Discuss this article in the C# forum.

**Read More**

**VB0209QA_T** Read this article online.

**VB0102JF_T** "Drill Down on VB.NET" by James Foxall

**VB5, VB6 • Intercept and Adjust Window Movements**

```
Option Explicit

Private Declare Sub CopyMemory Lib "kernel32" _
  Alias "RtlMoveMemory" (Destination As Any, _
  Source As Any, ByVal Length As Long)

Private Const WM_MOVING As Long = &H216

Private Type RECT
  Left As Long
  Top As Long
  Right As Long
  Bottom As Long
End Type

' ******************************************
'  Subclassing
' ******************************************
Friend Function WindowProc(hWnd As Long, _
  Msg As Long, wParam As Long, lParam As Long) _
  As Long
  Dim Result As Long
  Dim r As RECT
  Dim dX As Long
  Dim dY As Long

  ' Precalculate screen dimensions.
  dX = Screen.Width \ Screen.TwipsPerPixelX
  dY = Screen.Height \ Screen.TwipsPerPixelY

  Select Case Msg
    Case WM_MOVING
      ' Grab screen coordinates of drag rectangle.
      Call CopyMemory(r, ByVal lParam, _
        Len(r))
      ' Adjust to prevent window from going offscreen.
      If r.Left < 0 Then
        r.Right = r.Right - r.Left
        r.Left = 0
      End If
      If r.Top < 0 Then
        r.Bottom = r.Bottom - r.Top
        r.Top = 0
      End If
      If r.Right > dX Then
        r.Left = dX - (r.Right - r.Left)
        r.Right = dX
      End If
      If r.Bottom > dY Then
        r.Top = dY - (r.Bottom - r.Top)
        r.Bottom = dY
      End If
      ' Update drag rectangle for Windows.
      Call CopyMemory(ByVal lParam, r, Len(r))
      ' Let Windows know we've handled this.
      Result = True

    Case Else
      ' Pass along to default window procedure.
      Result = InvokeWindowProc(hWnd, Msg, _
        wParam, lParam)
  End Select

  ' Return desired result code to Windows.
  WindowProc = Result
End Function
```

**Listing 1** You can alter the position displayed as the user drags a window about the screen. Intercept the WM_MOVING message and alter the contents of the rectangle structure used by Windows to position the window's drag rectangle.

complex pointer arithmetic. This is probably why VB.NET doesn't support unsafe code.

Unsafe code also comes in handy when interoperating with Win32 API calls that require pointers. C# unsafe code uses C-like pointer syntax for the most part. You can only use unsafe code at a method's scope by prefixing the method definition with the reserved word "unsafe," then using C-pointer syntax for direct memory manipulation:

```
unsafe public void
  MyUnsafeMethod1(int*
  ptr)
{
  Debug.Assert(ptr != null);
  *ptr = 3;
}
```

You can only apply the unsafe qualifier to methods and properties, not to individual statements or class member variables. Note one important programming detail when dealing with unsafe code: You must pin down the memory sections you interact with directly, using a fixed statement, because garbage collection can start at any moment and move objects around in memory. The fixed statement takes this form:

```
fixed(type* ptr = expr)
```

It pins down the object ptr points at, while the expression in the statement executes:

```
unsafe void UnsafeArrayAccess()
{
  int[] intArray = new int[3];
  fixed(int* ptr = intArray)
  {
    *ptr     = 1;
    *(ptr+1) = 2;
    *(ptr+2) = 3;
    *(ptr+3) = 4;//compiles, but
unsafe and may cause error
  }
}
```

You don't need to pin down unsafe access to value types because value types are stack allocated, so they aren't subjected to garbage collection:

```
struct Point
{
  public int x;
  public int y;
}
//using unsafe struct
unsafe void UnsafeStructUsage()
{
  Point point;
  point.x = 1;
  point.y = 2;

  Point* pPoint = &point;
  pPoint->x = 3;
  pPoint->y = 4;
}
```

The C# compiler doesn't support unsafe code by default—you must enable it explicitly in your project configuration. In the Project Properties page, select Build, and set the "Allow unsafe code blocks" dropdown box to True (see Figure 1). —*J.L.*

## Q: Understand Checked and Unchecked Code

What is checked code? How is it different from normal C# code? Is it the same as managed code?

## A:

By default, the C# compiler and the CLR don't check for overflow or underflow after performing arithmetic operations. This is called unchecked code. As a result, you might get erroneous results without knowing it, even though it's valid managed code. For example, consider the CalcPower() method that returns the result of a specified number raised to a specified power:

```
int CalcPower (int num,int power)
{
   int result = 1;
   for(int i = 1;i<=power;i++)
   {
      result *= num;
   }
   return result;
}
```

Because int is only 32 bits long, trying to calculate CalcPower-(10,11) returns the bogus result of 1,215,752,192 instead of 100,000,000,000. You can instruct the C# compiler to throw an exception of type OverflowException in case of an overflow error, using the checked instruction:

```
int CalcPower(int num,int power)
{
   int result = 1;
   for(int i = 1;i<=power;i++)
   {
      checked
      {
         result *= num;
      }
   }
   return result;
}
```

Now, the calling client is aware that an error took place. Similarly, you can flag a code segment as explicitly unchecked using the unchecked instruction:

```
unchecked
{
   //some code
}
```

You can nest checked or unchecked statements inside each other. By default, the C# compiler generates unchecked code. To enable support for checked code, open the Project Properties page, select Build, and set "Check for Arithmetic Overflow/Underflow" to True (see Figure 1). You should use checked code for the "usual suspects"—that is, calculating powers, calculating factorials, and so on. —*J.L.*

## Q: Declare Constant Object Variables

I wrote a class and now I want to declare a constant instance of that class, but the VB.NET compiler won't let me. Is there a workaround?

## A:

The only types you're allowed to use in a VB.NET Const statement are the ones you can write literals for. These are the primitive types (numeric types, Boolean, Char, Date), String, and Object. The only value you can initialize an Object constant to is Nothing, because no allocation can occur.

The same rules apply to C#, except it doesn't provide a way to write date literals, so it doesn't support DateTime constants. On the other hand, it allows you to declare a null constant of any reference type, not only Object.

Fortunately, VB.NET has another modifier keyword, ReadOnly, which you can apply to a field variable declaration to make it behave almost like a constant. You can initialize a read-only variable where it's declared or in a constructor, but you're not allowed to change it after that. The compiler enforces this rule, so you get a compile error if you try to modify a variable marked as ReadOnly in a method.

Take a look at a couple different ReadOnly variables in use (see Listing 2). The code illustrates an important difference compared to constants. When you use a read-only variable, you can change how it's initialized based on calculations and parameters passed to the constructor. You're not limited to a single static value the way you are when declaring a constant.

Another significant distinction between the two is related to versioning. When you use a constant, its value is embedded in the code everywhere it's used. The CLR never refers to the constant field at run time, and in fact doesn't even load it into memory. This wouldn't work for a read-only field, because it's only semi-constant, and the VB.NET compiler can't determine its value at compile time.

**VB.NET • Compare Const With ReadOnly**

```
Class Ball

  Private Const PI As Double = 3.14159265

  ' The following doesn't work
  ' Public Const SmallBall As New Ball( 5 )

  Public Shared ReadOnly SmallBall As New Ball _
    ( 5 )
  Public Shared ReadOnly BigBall As New Ball _
    ( 20 )

  Public ReadOnly Radius As Integer
  Public ReadOnly Volume As Double

  Public Sub New(ByVal radius As Integer)
    Me.Radius = radius
    Volume = 4 / 3 * PI * radius^3
  End Sub

End Class
```

**Listing 2** You use constants for only simple, static values. The ReadOnly keyword offers more flexibility because it allows you to initialize the variable in a constructor based on calculations and parameters.

Because of this, you should only use constants for values you're certain will never change. If you change a constant, you'd have to recompile all code that uses it for the change to take effect.

Also note that the VB.NET Const keyword implies Shared, meaning you can always access the constant without having to instantiate the class. ReadOnly doesn't do this, so if you want a read-only field to be Shared, you must add the Shared keyword explicitly. —*M.S.*


**Karl E. Peterson** is a GIS analyst with a regional transportation planning agency and serves as a member of the *Visual Studio Magazine* Technical Review and Editorial Advisory Boards. Online, he's a Microsoft MVP and a section leader on several DevX forums. Find more of Karl's VB samples at www.mvps.org/vb.

**Juval Löwy** is the principal of IDesign, a consulting and training company focused on .NET design and migration. Juval is the author of *COM and .NET Component Services* (O'Reilly), and he speaks frequently at major conferences. Juval chairs the program committee of the .NET California Bay Area User Group. Contact him at www.idesign.net.


**Mattias Sjögren** lives in southern Sweden, where he tries to combine consulting work with university studies. He is a Microsoft MVP for Visual Basic. Reach him at mattias@mvps.org or visit his Web site at www.msjogren.net.


## Additional Resources

- HookMe.zip: www.mvps.org/vb/samples.htm
- "Read-Only Variables": http://msdn.microsoft.com/library/en-us/vbls7/html/vblrfvbspec7_5_2.asp
- "App Object Changes in Visual Basic .NET": http://msdn.microsoft.com/library/en-us/vbcon/html/vxconchangestoappobjectinvisualbasicnet.asp