Lazy Object / Weak Reference

Posted on September 11, 2018 by Rubberduck VBA

Sometimes a class needs to hold a reference to the object that "owns" it – i.e. the object that created it. When this happens, the *owner* object often needs to hold a reference to all the "child" objects it creates. If we say `Class1` is the "parent" and `Class2` is the "child", we get something like this:

```
1  'Class1
2  Option Explicit
3  Private children As VBA.Collection
4
5  Public Sub Add(ByVal child As Class2)
6      Set child.Owner = Me
7      children.Add child
8  End Sub
9
10 Private Sub Class_Initialize()
11     Set children = New VBA.Collection
12 End Sub
13
14 Private Sub Class_Terminate()
15     Debug.Print TypeName(Me) & " is terminating"
16 End Sub
```

And `Class2` might look like this:

```
1  'Class2
2  Option Explicit
3  Private parent As Class1
4
5  Public Property Get Owner() As Class1
6      Set Owner = parent
7  End Property
8
9  Public Property Set Owner(ByVal value As Class1)
10     Set parent = value
11 End Property
12
13 Private Sub Class_Terminate()
14     Debug.Print TypeName(Me) & " is terminating"
15 End Sub
```

The problem might not be immediately apparent to untrained eyes, but this is a *memory leak* bug – this code produces no debug output, despite the `Class_Terminate` handlers:

```
'Module1
Option Explicit

Public Sub Test()
    Dim foo As Class1
    Set foo = New Class1
    foo.Add New Class2
    Set foo = Nothing
End Sub
```

Both objects remain in memory and outlive the `Test` procedure scope! Depending on what the code does, this could easily go from "accidental sloppy object management" to a serious bug leaving a ghost process running, with Task Manager being the only way to kill it! How do we fix this?

Not keeping a reference to `Class1` in `Class2` would fix it, but then `Class2` might not be working properly. Surely there's another way.

Suppose we abstract away the very notion of *holding a reference to an object*. Suppose we don't hold an object reference anymore, instead we hold a `Long` integer that represents *the address at which we'll find the object pointer we're referencing*. To put it in simpler words, instead of holding the object itself, we hold a ticket that tells us where to go find it when we need to use it. We can do this in VBA.

First we define an interface that encapsulates the idea of an object reference – `IWeakReference`, that simply exposes an `Object` get-only property:

```
'@Description("Describes an object that holds the address of a pointer to an
'@Interface
Option Explicit

'@Description("Gets the object at the held pointer address.")
Public Property Get Object() As Object
End Property
```

Then we implement it with a `WeakReference` class. The trick is to use `CopyMemory` from the Win32 API to take the bytes at a given address and copy them into an object reference we can use and return.

For an easy-to-use API, we give the class a *default instance* by toggling the `VB_PredeclaredId` attribute, and use a *factory method* to create and return an `IWeakReference` given any object reference: we take the object's *object pointer* using the `ObjPtr` function, store/encapsulate that pointer address into a private instance field, and implement the `IWeakReference.Object` getter such that if anything goes wrong, we return `Nothing` instead of bubbling a run-time error.

```
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "WeakReference"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit
Implements IWeakReference

#If Win64 Then
Private Declare PtrSafe Sub CopyMemory Lib "kernel32.dll" Alias "RtlMoveMem
```

```
#Else
Private Declare Sub CopyMemory Lib "kernel32.dll" Alias "RtlMoveMemory" (hp
#End If

Private Type TReference
#If VBA7 Then
    Address As LongPtr
#Else
    Address As Long
#End If
End Type

Private this As TReference

'@Description("Default instance factory method.")
Public Function Create(ByVal instance As Object) As IWeakReference
    With New WeakReference
        .Address = ObjPtr(instance)
        Set Create = .Self
    End With
End Function

Public Property Get Self() As IWeakReference
    Set Self = Me
End Property

#If VBA7 Then
Public Property Get Address() As LongPtr
#Else
Public Property Get Address() As Long
#End If
    Address = this.Address
End Property

#If VBA7 Then
Public Property Let Address(ByVal Value As LongPtr)
#Else
Public Property Let Address(ByVal Value As Long)
#End If
    this.Address = Value
End Property

Private Property Get IWeakReference_Object() As Object
' Based on Bruce McKinney's code for getting an Object from the object poin

#If VBA7 Then
    Dim pointerSize As LongPtr
#Else
    Dim pointerSize As Long
#End If

    On Error GoTo CleanFail
    pointerSize = LenB(this.Address)

    Dim obj As Object
    CopyMemory obj, this.Address, pointerSize

    Set IWeakReference_Object = obj
    CopyMemory obj, 0&, pointerSize

CleanExit:
```

```
76          Exit Property
77
78      CleanFail:
79          Set IWeakReference_Object = Nothing
80          Resume CleanExit
81      End Property
```

Now `Class2` can hold an *indirect* reference to `Class1`, like this:

```
1   'Class2
2   Option Explicit
3   Private parent As IWeakReference
4
5   Public Property Get Owner() As Class1
6       Set Owner = parent.Object
7   End Property
8
9   Public Property Set Owner(ByVal Value As Class1)
10      Set parent = WeakReference.Create(Value)
11  End Property
12
13  Private Sub Class_Terminate()
14      Debug.Print TypeName(Me) & " is terminating"
15  End Sub
```

Now `Module1.Test` produces the expected output, and the memory leak is fixed:

```
Class1 is terminating
Class2 is terminating
```

Posted in OOP, tutorials, vbaTagged design-patterns, memory-leak, oop, tutorial, vba

Published by Rubberduck VBA

I'm Mathieu Guindon (Microsoft MVP Office Apps & Services, 2018), you may have known me as "Mat's Mug" on Stack Overflow and Code Review Stack Exchange. I manage the Rubberduck open-source project, whose goal is to bring the Visual Basic Editor (VBE) - VBA's IDE - into the 21st century, by providing features modern IDE's provide. *View all posts by Rubberduck VBA*

18 thoughts on "Lazy Object / Weak Reference"

1. **A** *September 11, 2018* *Reply*

Good article!
But please tell me, why do you on the one hand use "#If Win64" and on the other "#If VBA7" to distinguish between "Long" and "LongPtr"?

Rubberduck VBA *September 11, 2018* *Reply*

Because VBA6 didn't have a LongPtr type =)

Rubberduck VBA *September 11, 2018* *Reply*

TBH the conditional compilation isn't bullet-proof here; a 64-bit Windows running VBA6 probably wouldn't be able to compile it, precisely because of the "#If Win64" check you're highlighting (PtrSafe only exists in VBA7). Anyway the idea is to make sure we're using the correct size for the object pointer; hard-coding a Long wouldn't work with all possible configurations.

2. **Eric van Rooijen** *September 11, 2018* *Reply*

Just a thought; because your collection (children) has a link/reference to Class2, the terminate events won't be fired. They will be fired when you first set your collection to Nothing before setting foo to Nothing

Rubberduck VBA *September 11, 2018* *Reply*

In this example, yes. I'll triple-check again, but the reason I wrote this (perhaps easily abused) class was precisely because of a scenario where nulling the parent reference in the child instance, did not untie the knot; that case was an event provider custom class wrapping a dynamic MSForm control, with the form having a private WithEvents reference to the child class. I took it that explicitly nulling references wasn't consistently reliable and proceeded to make the object references indirect like this, and the problem was resolved – I'll probably make a follow-up post, or edit this one later, when I have a more "real-world" piece of example code for it… on the other hand, making the reference explicitly "weak" removes the need to explicitly set it to Nothing, so.. I'm not sure whether to consider it abuse or over-complexifying the situation, vs. making such intertwined references safe to use without needing to think of nulling them.

3. **Profex** *September 19, 2018* *Reply*

FYI, the only Win32 API functions (of all the ones that I use) that require "#If Win64" are "GetWindowLong" & "SetWindowLong" (within the "#If VBA7" condition). Otherwise, "#If VBA7" is the main difference because it introduced "LongPtr".

4. **Profex** *September 19, 2018* *Reply*

Q. What is the difference between using 'CopyMemory obj, 0&, pointerSize" or replacing it with with "Set obj = nothing"?…and why does it crash Excel (after class1 terminates)?

Rubberduck VBA *September 19, 2018* *Reply*

Interesting – I haven't experienced any crashes with the code exactly as it is, on both 32 and 64 bit hosts, although I did toy a bit with that code: omitting the last CopyMemory call immediately crashes everything. I figured there was a reason Bruce McKinney did it that way, so I left it alone – wouldn't "Set obj = Nothing" confuse the reference-counting though? The

object wasn't created by normal means, it shouldn't be destroyed by normal means either. Are you saying *this code* crashes? What version+bitness of Excel are you using, in what OS+bitness? Or is it "Set obj = Nothing" that's crashing?

1. **Profex** *September 19, 2018*

   Using "Set obj = Nothing" crashed Excel every time, but Bruce's method was fine. I'm using Win 7(64bit)/Excel 2013(32-bit).

   I had to add a few lines of code, so that I could access the "Owner" property of the child, to actually trigger "IWeakReference_Object" to run. This included a "Property Get Item(ByVal Index As Long) As Object" in Class1.

5. **SmileyFtW** *October 9, 2018* *Reply*

   Explains memory issues I have experienced… will be using this technique going forward. Thank you, Mathieu!

6. **Beryl Hesh** *October 17, 2018* *Reply*

   Tried using this with a reference to a Workbook in a class that is cached, in order to avoid a phantom VBE reference hanging around when the Wb is closed. It works great for that purpose, but a subsequent test of the reference leads to a crash instead of a clean failure. Any suggestions? Am using 64-bit Office 365.

   Rubberduck VBA *October 17, 2018* *Reply*

   Hmm, I intended this to use with custom VBA classes, which don't involve COM objects that are owned by the host application… there is likely something else going on here. I'd be curious to see the original code with the ghost instance.. are you working in Excel or creating an Excel.Application object? Are you accessing VBE objects? Is the VBE Extensibility library referenced? It's very easy to make "ghost" objects with chained member calls under these circumstances. With a good MCVE, that would make a great question on Stack Overflow!

7. **Beryl Hesh** *October 17, 2018* *Reply*

   I made some dummy classes and test cases that illustrate the problem. Will be a detailed SO question, but I agree an interesting one, so will post one at some point later. What does MCVE stand for though??

8. **Beryl Hesh** *October 17, 2018* *Reply*

   Never mind, gOOgle just told me 🙂

9. **markjohnstoneblog** *December 11, 2018* *Reply*

   Great article and very interesting topic regarding memory leaks and the deconstructor Class_Terminate() not firing. Will have to dig up where I read that with Preclared classes you require to write your own deconstructor to do the cleanup. The Lazy Object/Weak Reference seems a bit of "hack" to avoid memory leaks thou an important topic that's easily overlooked. On that note, I'm sure my memory requires some cleaning up as keep forgetting things. 😀

10. **SmileyFtW** *February 8, 2019* *Reply*

Wondering if combining this topic with ideas you share elsewhere (“There Is No Workbook”, I think) is something worth doing or if there are pitfalls in it. Also you mentioned above that this is “easily abused” – How so? You also said you might revisit this and provide a more “real world” version…
Here’s the code before the change I’m considering:
‘Class2
Option Explicit
Private parent As Class1
Public Property Get Owner() As Class1
Set Owner = parent
End Property

And then after:
‘Class2
Option Explicit
Private Type TModel
Parent As Class1
End Type
Private this as TModel
Public Property Get Parent() As Class1
Set Parent = this.Parent
End Property

1. **SmileyFtW** *February 8, 2019* *Reply*

   It isn’t the “There Is No Workbook” post it’s the “Apply Logic” (https://rubberduckvba.wordpress.com/2018/05/08/apply-logic-for-userform-dialog/)…

2. **SmileyFtW** *February 9, 2019* *Reply*

   And “Set Parent = this.Parent” should have been “Set Parent = this.Parent.object”