Coercion Aversion

In the new version of Visual Basic, watch out for ETC, unaffectionately known as "Evil Type Coercion."

by Karl E. Peterson

One of the more interesting topics of discussion during the long beta process of VB4 was Evil Type Coercion, or ETC as it was more commonly referred to. While one of Basic's strengths as a language has always been that it could coerce one type of variable to be another, the rules followed for such coercions were generally well understood and very strict. For instance, if you assigned an Integer value to a Long variable, Basic wouldn't throw an error but would simply coerce the data from one type to another. But, if you assigned a String to an Integer, kaboom! The result was highly predictable—a Type Mismatch Error.

The rules have changed with VB4. Now, if it's even remotely possible, your data will be coerced to fit into whatever variable type is thrown at it. Suppose you have a routine that expects to receive a String and an Integer. (For this test, it's not important what the routine does, just what shape the parameters arrive in.) The routine calls the String function to generate a string of *aStr* characters that is *aNum* long, and this is displayed in the Debug window:

```
Sub ETCDemo1(aStr As String, aNum As Integer)
   Debug.Print String(aNum, aStr)
End Sub
```

Say you accidentally reverse the parameters in the call, passing the Integer first and the String second. VB3 would throw a "Type Mismatch" error, right? A simple test shows that not only does VB4 fail to generate an error, but it happily coerces your parameters to fit:

```
Sub Main()
   Dim i As Integer
   i = 4
   Call ETCDemo1(i * 2, CStr(i))
   '<-- Oops! Switched parameters
   Call ETCDemo1(CStr(i), i * 2)
   '<-- Parameters in proper order
End Sub
```

The first call to ETCDemo1 prints "8888" in the Debug window. Welcome to ETC! Were the parameters not (accidentally) reversed, as the second call shows, the intended result would have been "44444444." The "rule" in this case is that if you pass a variable by value (this includes expressions, temporary variables, and literals), and it can be coerced into what the called routine expects, it will be. This is totally new behavior that no version of Basic has ever exhibited, so it could easily catch you off guard. And, even though you might expect it would help, Option Explicit has no impact on this bizarre behavior. Watch out for it.

NEW MATH: A+B<>1+A+B???

ETC can also lead to some very strange "math" errors. I'll add another demo routine to show you just how messed up this can get:

```
Sub ETCDemo2(a As String, b As String)
   Debug.Print "a = "; a, , TypeName(a)
   Debug.Print "b = "; b, , TypeName(b)
   Debug.Print "a + b = "; a + b, , _
      TypeName(a + b)
   Debug.Print "a + 1 = "; a + 1, , _
      TypeName(a + 1)
   Debug.Print "a + b - 1 ="; a + b - 1, _
      TypeName(a + b - 1)
   Debug.Print "1 - a + b ="; 1 - a + b, _
      TypeName(1 - a + b)
   Debug.Print "a + b + 1 ="; a + b + 1, _
      TypeName(a + b + 1)
   Debug.Print "1 + a + b ="; 1 + a + b, _
      TypeName(1 + a + b)
End Sub
```

Call ETCDemo2 passing the literals 1 and 23 as the first and second parameters:

```
Call ETCDemo2(1, 23)
```

This output appears in the Debug window:

```
a = 1                    String
b = 23                   String
a + b = 123              String
a + 1 = 2                Double
a + b - 1 = 122          Double
1 - a + b = 23           Double
a + b + 1 = 124          Double
1 + a + b = 25           Double
```

The first thing you'll probably notice is that VB4 didn't throw an error when you passed numbers into a routine that

Karl E. Peterson is a GIS Analyst with a regional transportation planning agency and a member of the Visual Basic Programmer's Journal *Technical Review Board. He's also an independent programming consultant and a writer based in Vancouver, Washington. He is the coauthor of* Visual Basic 4.0 How To*, from Waite Group Press. He's the 32-Bit Bucket Section Leader for the* VBPJ *Forum and a Microsoft MVP in the MSBASIC Forum. Contact Karl in either CompuServe location at 72302,3707.*

was expecting strings. This is disturbing enough. But, now look at "new math," as VB4 defines it. The plus sign in Basic has historically served a dual purpose of string concatenation and numeric addition, so it makes sense that (a + b) is "123." However, VB3 would have immediately thrown an error on the (a + 1) instruction, but VB4 merrily coerces "1" to 1 and performs the addition.

Worse, there has never been an affinity between the minus sign and strings! Yet, we can subtract 1 from "123" and return a numeric value of 122. Completely independent of your instruction, VB4 has decided that you *really* meant for that string to be a number. Likewise, if you subtract "1" from 1, then add "23," you end up with a numeric 23. The so-called logic here is that coercion precedence is occurring left to right, with VB4 favoring a numeric result over a string when one operand is a number.

This brings us to the title of this topic. By now, it should make total sense to you, right? (Please excuse the sarcasm. <g>) Using our left-to-right precedence rule, we can say that the concatenated string (a + b) will be "123" and adding 1 will coerce the "123" to 123 for a result of 124. Placing the 1 at the front of the equation changes the coercion order, though. Now, 1 plus "1" is 2, and "23" added to that will result in 25. Absolutely logical, right? (Sorry, there's that sarcasm again. <g>)

The last thing I'd like point out from this example is just how "optimized" these coercions make our calculations. Note that the results, intermediate and final, of the calculations are double precision. In other words, if you accidentally introduce a string into your tight integer loop, you've greatly increased the overhead of the math.

If it bothers you that VB now treats specifically typed variables as Variants, you may wish to contact Microsoft and express your displeasure. Perhaps rigorous type checking can be reinstituted in VB5 if enough folks really care about this. In the meantime, you can take some alternate routes to avoid such ETCs. First and foremost, you must make absolutely sure that you don't introduce strings into equations. Second, pass variables by reference rather than by value. This is more than a little ironic because, historically, passing variables by value was safer than passing them by reference. You may also want to consider adopting the ampersand notation for concatenating strings. Because this newer operator doesn't carry the historic baggage of the plus sign, and was introduced with coercion fully documented, you're much less likely to accidentally trip over this "new math." Be as type-explicit as possible, and there may be some hope. (My thanks to Gregg Irwin and Zane Thomas for their examples of the ETC problems, and for helping me carry the torch on this issue!)

Now that I've voiced my opinion on the ETC issue, I'll move on to a very cool, barely documented, design feature in VB4. You can now adjust the position or size of controls with your cursor keys. To move a control, select it, then hold the Control key down while pressing the arrow keys. Similarly, holding down the Shift key while pressing the arrow keys resizes the control. This trick works just as well on multiple controls if you want to move or size them as a group. If you have Align Controls to Grid checked on the Environment dialog (select Tools, Options, then Environment), each press of an arrow key will increment the position or size by your grid spacing. If this option is unchecked, the control(s) will move or size by one pixel with each keystroke.

BUILDING A "TYPEAMATIC" LIST-BOX SEARCH

If you've played much with the Windows 95 Explorer utility, or

VB4

```
Option Explicit
'
' API Declarations
'
#If Win16 Then
  Private Declare Function SendMessage Lib _
     "User" (ByVal hWnd As Integer, ByVal _
     wMsg As Integer, ByVal wParam As _
     Integer, lParam As Any) As Long
  Private Const LB_FINDSTRING = &H410
#ElseIf Win32 Then
  Private Declare Function SendMessage Lib _
     "user32" Alias "SendMessageA" (ByVal _
     hWnd As Long, ByVal wMsg As Long, _
     ByVal wParam As Long, lParam As Any) _
     As Long
  Private Const LB_FINDSTRING = &H18F
#End If
Private Const LB_ERR = (-1)
'
' Flag to indicate if keystrokes are
' being accepted.
'
Private ExtendingSearch As Boolean
'
' Time last keystroke was entered.
'
Private LastKey As Double
'
' Timer settings
'
Private Const msTimeLimit = 2000  '2 seconds
Private Const msCheckEvery = 200  '1/5 second

Private Sub Command1_Click()
  Unload Me
End Sub

Private Sub Form_Load()
  Dim file As String
  '
  ' Fill list box with filenames.
  '
  file = Dir(Environ("windir") & "\*.*")
  Do While Len(file)
     List1.AddItem file
     file = Dir()
  Loop
  '
  ' Set timer interval and initialize vars.
  '
  Timer1.Interval = msCheckEvery
  LastKey = Now
End Sub

Private Sub List1_GotFocus()
  '
  ' Restart timer and clear search string.
  '
  Timer1.Enabled = True
  ExtendingSearch = False
End Sub

Private Sub List1_KeyPress(KeyAscii As _
  Integer)
  Static Search As String
```

CONTINUED ON NEXT PAGE.

LISTING 1 ***Building a Searching List Box.*** *You can use the LB_FINDSTRING message to locate list items that start with a given string. Together with a timer, you can build functionality similar to that in the Windows 95 Explorer. Place this code in a VB4 form with a list box, timer, label, and command button.*

with Norton Desktop for Windows, you've probably noticed the neat way you can search for files by typing the first few letters while the list box has focus. This is accomplished by sending an LB_FINDSTRING message to the list box with each keystroke the user enters. One method used to measure the time between keystrokes is with a Timer control. If more than a set amount of time elapses, the search string is reset so a new search can begin. Because a number of events must work together to make this technique work, I've listed the entire form's code (see Listing 1). You can download the whole project from the Magazine library of the *VBPJ* Forum on CompuServe (search for PT1195.ZIP).

I implemented this as a conditionally compiled 16/32-bit form, so it can run in either version of VB4. The only things I had to think about were the declaration for SendMessage and the LB_FINDSTRING constant; everything else worked out naturally. To run this in VB3, remove the Private keywords (and replace with Dim, if necessary), change the Boolean to Integer, and remove the conditional compilation directives and 32-bit declares.

Other than the API declarations, two Private (form-level) variables are declared to serve as status trackers that must be available to multiple events within the form. A form-level Boolean variable, ExtendingSearch, indicates whether the current search string should have new characters appended to it, or if it should be cleared and a new search begun when the user enters another keystroke. Another form-level Double variable, LastKey, stores the time when the user entered the last keystroke.

The form itself is very simple (see Figure 1). It has a list box, command button, label, and timer on it. For these controls, the only setting that's really critical is the Sorted property of the list box. While the demo would still work, so to speak, it would appear odd at best if this property were not set to True. During the Form_Load event, the list box is filled with the contents of your Windows directory. Also during the Form_Load event, the timer Interval property is set according to a constant value defined in the Declarations section, and the time of the last keystroke is set to Now. As for the other controls, the command button's only job is to unload the form, and the label displays which keystrokes are being used to perform the searches.

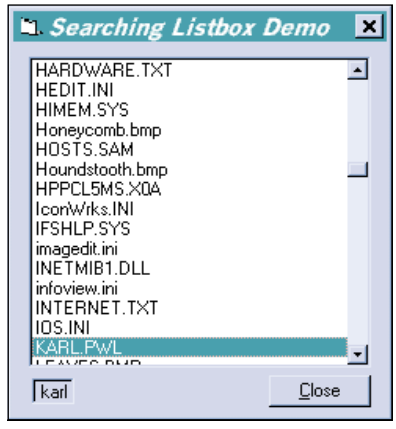

FIGURE 1 ***The Searching List Box in Action.*** *This VB4 project demonstrates a simple use of SendMessage to find the first string in a list that matches whatever the user types without pausing for more than two seconds between keystrokes.*

The LB_FINDSTRING message searches a list box for the first

LISTING 1 CONTINUED FROM PREVIOUS PAGE.

```
    Dim Index As Long
    Dim DoSearch As Boolean
    '
    ' Start over if delay was too long.
    '
    If Not ExtendingSearch Then
       Search = ""
       ExtendingSearch = True
       Index = List1.ListIndex
    Else
       Index = -1
    End If
    '
    ' Check for valid keystrokes.
    '
    If KeyAscii = vbKeyBack Then
       '
       ' Allow user to take back last key.
       '
       If Len(Search) Then
          Search = Left(Search, Len(Search) - 1)
          Label1 = " " & Search & " "
          If Len(Search) Then
             DoSearch = True
          Else
             DoSearch = False
          End If
       End If
    ElseIf KeyAscii >= vbKeySpace Then
       '
       ' Append latest key.
       '
       Search = Search & Chr(KeyAscii)
       Label1 = " " & Search & " "
       DoSearch = True
    End If
    '
    ' Perform search after valid keystrokes.
    '
    If DoSearch Then
       Index = SendMessage(List1.hWnd, _
          LB_FINDSTRING, Index, ByVal Search)
       If Index <> LB_ERR Then 'Found a match!
          List1.ListIndex = Index
       Else 'No match
          Search = Left(Search, Len(Search) - 1)
          Beep
          Label1 = " " & Search & " "
       End If
       '
       ' Record when key was pressed, and consume
       ' keystroke so VB doesn't automatically
    ' move list to entry that starts with
       ' last key.
       '
       LastKey = Now
       KeyAscii = 0
    End If
End Sub

Private Sub List1_LostFocus()
    '
    ' Turn off timer for efficiency.
    '
    Timer1.Enabled = False
End Sub

Private Sub Timer1_Timer()
    Dim Elapsed As Double
    '
    ' Check if more than allowed time has     ' elapsed.
    '
    If ExtendingSearch Then
       Elapsed = Now - LastKey
       If (Elapsed * 86400) > (msTimeLimit / _
          1000) Then
          ExtendingSearch = False
          Label1 = Label1.Tag
       End If
    End If
End Sub
```

entry that *starts with* the string passed in lParam. It begins its search at the index value passed in wParam, and will wrap around to the beginning of the list if a match is not found below the index entry. If -1 is used as the index, the search starts at the very beginning of the list.

The real work of this demo is performed in the List1_KeyPress event. First, code within the List1_KeyPress event determines if the last search should be extended by appending the new keystroke to the previous search string. (This search string is held as a static variable within the KeyPress event.) If so, the program assigns the search's start index from the beginning of the list through SendMessage to send the LB_FINDSTRING message to the list box.

If the keystroke were a backspace, you would use the Left function to trim the last character from the previous search string. If the keystroke were any alphanumeric key, generally one whose ASCII value is greater than or equal to the spacebar, you would append the character it represents to the previous search string. In both cases, you update the label control to show the user what string is about to be used for the search.

Now, you're ready to actually call SendMessage. With this message, a return value of LB_ERR (-1) indicates failure. Any other return value is the index for the matching string that was found in the list box. If a match is found, you simply set the list index to that return value. If no match is found, write code that trims the most recent keystroke from the search string and beeps to inform the user of the failure. In both cases, record the time of the keystroke, and set the KeyAscii value to zero. If the KeyAscii value is not set to zero, VB will move to the first item in the list that starts with the last letter the user entered after the KeyPress event is exited.

The last element in the technique is within the Timer event. Here, if ExtendingSearch is True, the elapsed time since the last keystroke took place is compared with a constant value (two seconds, in this case) to determine if the current search should continue to be extended or whether a new search should begin. To calculate the elapsed time, just subtract the time of the last keystroke from Now. Multiply this value by 86,400 (number of seconds in a day) to convert to seconds. If the time limit has expired, toggle the ExtendingSearch flag and clear the status label. For efficiency, the timer is disabled whenever the list box loses focus, and reenabled when the list box regains focus.

Use of the Timer is not mandatory. You can write this searching routine just as effectively without it, but the Timer does facilitate showing the user what the current search string is, and when the current search will no longer be extended. A modified version of this code that eliminates the timer is included in the Magazine library of the *VBPJ* Forum on CompuServe (search for PT1195.zip).

FINDING A SPECIFIC EXECUTABLE

My next tip was written for VB3, but the code will also run in the 16-bit version of VB4. To run the code in the 32-bit version of VB4, you will need new declarations and conditional compilation.

Many times, you don't know whether a given executable is already running, and would like to find out before attempting to communicate with it or starting a new instance. You can use a GetWindow loop to test the module name of each top-level window encountered (for more information on the GetWindow loop, see Programming Techniques, *VBPJ*, September 1995). This may be the only recourse when there is no reliable caption to search for, although I'd recommend it as a last resort because file names may change over time, and those

VB3

```
' Win16 Declarations
Declare Function ShowWindow Lib "User" _
  (ByVal hWnd As Integer, ByVal nCmdShow _
  As Integer) As Integer
Declare Function FindWindow Lib "User" _
  (ByVal lpClassName As Any, ByVal _
  lpWindowName As Any) As Integer
Declare Function GetParent Lib "User" _
  (ByVal hWnd%) As Integer
Declare Function GetWindow Lib "User" _
  (ByVal hWnd%, ByVal wCmd%) As Integer
Declare Function GetWindowWord Lib "User" _
  (ByVal hWnd%, ByVal nIndex%) As Integer
Declare Function GetModuleFileName Lib _
  "Kernel" (ByVal hModule As Integer, ByVal _
  lpFilename As String, ByVal nSize As _
  Integer) As Integer

' GetWindow and GetWindowWord Constants
Const GW_HWNDNEXT = 2
Const GWW_HINSTANCE = -6

Function GetModuleHWnd (ByVal Module$) As Integer
  Dim hWndTmp%
  Dim hInstTmp%
  Dim ModTmp$
  Dim nRet%
  '
  ' Clean up module name, and create buffer to
  ' receive names of running modules.
  '
  Module = Trim$(UCase$(Module))
  ModTmp = Space$(128)
  '
  ' Find first window and loop through all
  ' subsequent windows in master window list.
  '
  hWndTmp = FindWindow(0&, 0&)
  Do Until hWndTmp = 0
     '
     ' Make sure this window has no parent.
     '
     If GetParent(hWndTmp) = 0 Then
        '
        ' Retrieve the instance handle, and use
        ' that to retrieve the module name.
        '
        hInstTmp = GetWindowWord(hWndTmp, _
           GWW_HINSTANCE)
        nRet = GetModuleFileName(hInstTmp, ModTmp, 128)
        '
        ' Compare this window's module name to
        ' that passed in.  Exit if match found.
        '
        If Right$(Left$(ModTmp, nRet), _
           Len(Module)) = Module Then
           GetModuleHWnd = hWndTmp
           Exit Do
        End If
        Debug.Print Left$(ModTmp, nRet)
     End If
     '
     ' Get next window in master window list
     ' and continue.
     '
     hWndTmp = GetWindow(hWndTmp, GW_HWNDNEXT)
  Loop
End Function
```

LISTING 2 ***Finding a Window With Only a File Name.*** *The GetModuleHWnd function uses a GetWindow loop to find the first running application whose file name matches a file name you pass in.*

"pesky users" <g> may even rename files! Another somewhat fatal flaw in this strategy is that it is not totally reliable under Windows NT. But sometimes using a GetWindow loop is simply your only option.

The GetModuleHWnd function (see Listing 2) loops through all the windows in the system, until it finds a top-level (one with no parent) window whose module name matches that passed into it. A window may be identified as top-level using the GetParent API. If this is the case, you must then obtain its file name. The GetModuleFilename API function will return the file name for any given instance handle (hInst). But because this loop is through a list of hWnds, each window's handle must be converted to an instance handle for its respective application. The GetWindowWord API returns various information about a window, one option being its hInst.

Passing the hInst returned by GetWindowWord to GetModuleFilename fills a buffer with the fully qualified file name of the process. All that's required now is a case-insensitive comparison with the file name that was passed into GetModuleHWnd. By comparing only as many characters from the right as are in the test string, you offer the option of ignoring paths (by not fully qualifying the passed string), or making the path a part of the comparison. If a match is found, the function returns that window's hWnd to the calling routine. A sample applet, RUNNING.MAK, which finds, restores, and activates other running programs, is included in the Magazine library of the *VBPJ* Forum on CompuServe (search for PT0995.ZIP). ×

STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION REQUIRED BY 39 U.S.C. 3685

1. Visual Basic Programmer's Journal, Pub. No. 10751955

2. Date of Filing: September 25, 1995

3. Published monthly

3A. No. of issues published annually: 12

3B. Annual subscription price: $34.97

4. Known office of publication: 209 Hamilton Ave. Palo Alto, CA 94301-2500

5. Complete mailing address of the headquarters of the general business offices of the publisher: Fawcette Technical Publications Inc., 209 Hamilton Ave. Palo Alto, CA 94301-2500

6. The names and addresses of the Publisher, Editor, and Managing Editor are Publisher: James Fawcette, Fawcette Technical Publications Inc., 209 Hamilton Ave. Palo Alto, CA 94301-2500; Editor: James Fawcette, Fawcette Technical Publications Inc., 209 Hamilton Ave. Palo Alto, CA 94301-2500; Managing Editor: Christine McGeever, Fawcette Technical Publications Inc., 209 Hamilton Ave. Palo Alto, CA 94301-2500

7. The owner is: Fawcette Technical Publications Inc. 209 Hamilton Ave. Palo Alto, CA 94301-2500. The names and addresses of stockholders owning or holding 1% or more of the total stock are listed below:
James Fawcette, 209 Hamilton Ave. Palo Alto, CA 94301-2500.

8. There are no known bondholders, mortgages, or other securities.

10. Extent and nature of circulation
Average no. copies each issue during preceding 12 months: A. Total no. copies (net press run), 101,258; B. Paid and/or requested circulation 1. Sales through dealers and carriers, street vendors and counter sales, 20,782; 2. Mail subscription (paid and/or requested), 65,169; C. Total paid and/or Requested Circulation (Sum of 10B1 and 10B2), 85,951; D. Free distribution by mail carrier or other means, samples, complimentary, and other free copies, 4,075; E. Total distribution (Sum of C and D), 90,026; F. Copies not distributed 1. Office use, left over, unaccounted, spoiled after printing, 1,355; 2. Return from News Agents, 9,878; G. Total (Sum of E, F1, and 2 - should equal net press run shown in A), 101,259.
Actual no. copies of single issue published nearest to filing date: A. Total no. copies (net press run), 160,014; B. Paid and/or requested circulation 1. Sales through dealers and carriers, street vendors and counter sales, 31,669; 2. Mail subscription (paid and/or requested), 96,626; C. Total paid and/or Requested Circulation (Sum of 10B1 and 10B2), 128,295; D. Free distribution by mail carrier or other means, samples, complimentary, and other free copies, 12,443; E. Total distribution (Sum of C and D), 140,738; F. Copies not distributed 1. Office use, left over, unaccounted, spoiled after printing, 1,400; 2. Return from News Agents, 17,876; G. Total (Sum of E, F1, and 2 - should equal net press run shown in A), 160,014.

11. I certify that the statements made by me above are correct and complete.

James Fawcette, President

HOME