Visual FoxPro Tips and Tricks

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Website: www.tomorrowssolutionsllc.com
Email: tamar@tomorrowssolutionsllc.com

I love to watch other people work at their computers. Every user makes a unique set of choices about when to use the mouse and when to use the keyboard, whether to use the menu or a keyboard shortcut, and so forth. But people often get set in their ways and continue to do something one way when there's a much more efficient way to do it.

The same thing applies to Visual FoxPro. It's been said that if you can do something at all, you can do it three ways. But most of us learn how to do something and move on, even if a different way is more efficient or more effective. When new ways of doing things come along, we don't always pay attention to them.

In this session, we'll look at a variety of ways to work smarter in VFP. Some apply to the IDE, while others address the language. In the tradition of Drew Speedie, we'll cover as many of these tips and tricks as time permits.

Introduction

There's a story about a young woman who, whenever she cooks a roast, cuts off both ends. One day, her husband asks her why she does it. She replies "I don't know. My mother always did it this way." Curious, she picks up the phone and calls her mother and says "Mom, why do you always cut off the ends of a roast?" Her mother thinks for a moment and says "I don't actually know. My mother always did that." So the young woman calls her grandmother and asks "Why do you always cut off the ends of a roast?" Grandma replies instantly, "Because my pan is too small."

Many of us have developed all kinds of habits without really thinking about them. We learn how to do something; do it a few times, so that it becomes automatic; and move on. On the whole, that's a good strategy. If we had to think about how to do each thing we do in a day, we wouldn't accomplish much, and we'd be exhausted.

But the flip side of building habits this way is that we don't always notice faster or better ways to do things (why didn't Grandma buy a bigger pan or a smaller roast?), or if we see them once, we don't manage to convert them to habits.

The goal of this session is to show you lots of things that can save you time or make your code better, in hopes that you'll grab a few of them and add them to your habits. I've divided them up broadly according to whether they apply to the VFP IDE or to code you write, and then further divided them within those categories.

Be more efficient in the IDE

As developers, we spend our days working in the Integrated Development Environment (IDE) that includes the Command Window, the Form and Class Designers, code windows, the Property Sheet, the Debugger, and other tools. The more efficiently we can work there, the more we can get done.

The first way to be more efficient is to set things up the way you like them. Whether it's choosing colors that pop, making things readable, or giving yourself shortcuts, VFP offers lots of ways to make the IDE yours.

Set fonts and sizes

FoxPro has let you set the font, including size, for the windows of the IDE for a long time, though the mechanism has changed over time. Not only can you set the font for a wide variety of window types, but you can override that setting for an individual window and VFP will remember. In VFP 9, you can even override those overrides.

To set the default font/size for the different types of windows, choose Tools | Options from the menu and click the IDE tab. This page (shown in **Figure 1**) has a dropdown to choose

the window type. For each type, you can set the font as well as a number of other editing choices.

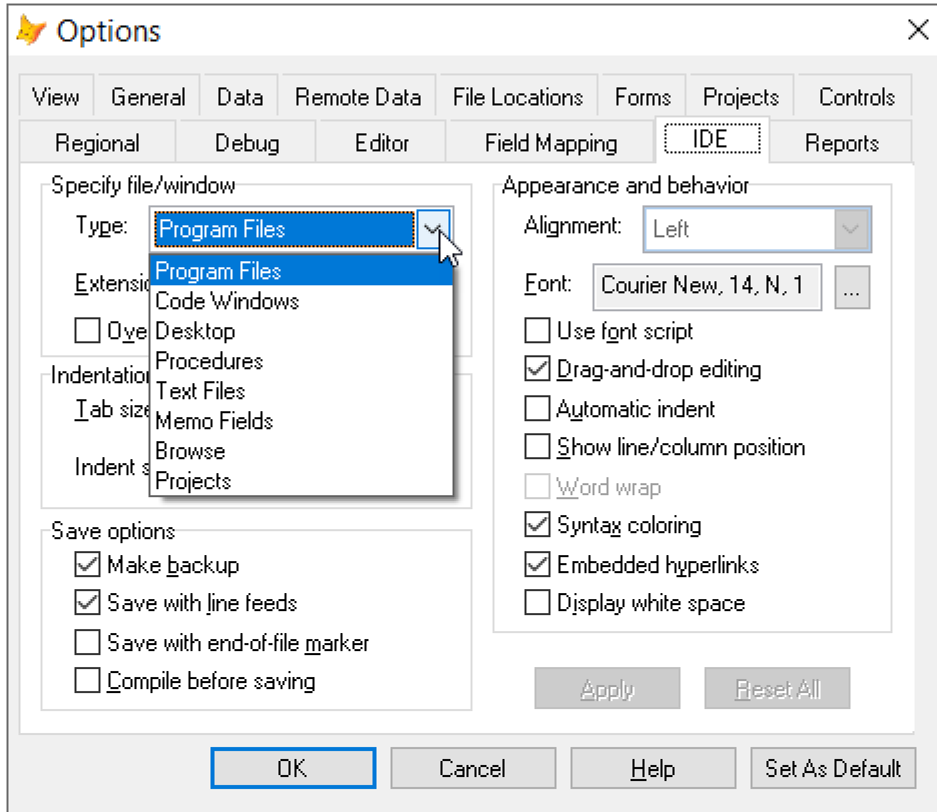

Figure 1. The IDE tab of the Options dialog lets you set default font and size for each type of window.

Once you've made any changes to the current window type, the Apply button is enabled. Click it to save your changes. If you choose another window type in the dropdown without applying your changes, VFP prompts you, as in **Figure 2**.

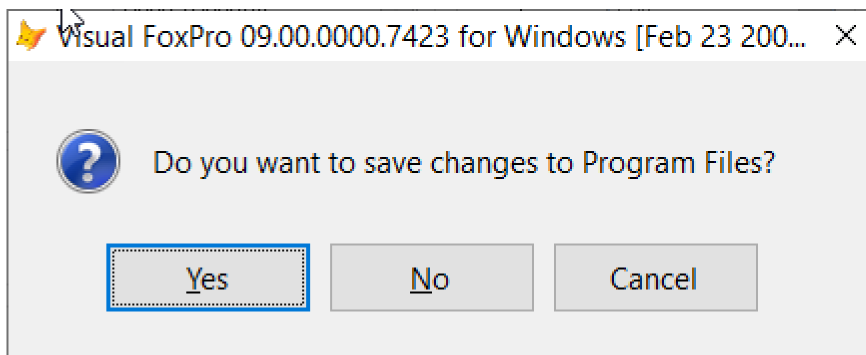

Figure 2. If you choose another window type with saving changes, VFP prompts you.

Table 1 shows what's covered by each item in the Type dropdown. Hidden below the dropdown in **Figure 1**, a textbox lets you specify which file types are included in the "Program Files" and "Text Files" categories.

Table 1. Each type in the dropdown on the IDE tab covers a different area of VFP.

Window type	Includes
Program Files	Code windows normally opened by MODIFY COMMAND. You can specify what extensions this type includes.
Code windows	Method editor windows opened by the Form and Class Designers.
Desktop	The VFP Desktop (background window).
Procedures	The code window for a database's stored procedures.
Text Files	Text windows normally opened by MODIFY FILE. You can specify what extensions this type includes.
Memo Fields	Memo editor windows normally opened by MODIFY MEMO or by double-clicking a memo field in a Browse window.
Browse	Browse windows.
Projects	The content inside the Project Manager. These settings do not apply to the Project Manager's tabs or buttons.

Each of these window types also includes a mechanism to change the settings for a particular window. (In most cases, you right-click and choose Properties. In the Project Manager, it's right-click, Font.) Changes you make to an individual window are stored in your resource file (FoxUser.DBF); they normally override the default settings. However, the IDE tab contains a checkbox labelled "Override individual settings" (partially hidden by the dropdown in **Figure 1**); when it's checked for a window type, all windows of that type use the editing settings you specify in the Options dialog.

It's worth spending some time experimenting with the other choices on the IDE page, too, to find the settings that make you most productive.

The list of types doesn't include all editing windows. To set the font for the Command Window, right-click there, choose Properties and set the desired font. I can't find a way to set default fonts for the Setup and Cleanup windows of the Menu Designer, but you can set them for a particular menu by right-clicking and choosing Properties.

The Property Sheet (technically, the Properties Window) is another that's not covered in the Options dialog. Like the Command Window, no doubt that's because there's just a single instance, so no need for default settings. To set the Property Sheet font, right-click somewhere below the title bar and no lower than the tabs, as in **Figure 3**, or in the description section at the bottom. Then choose Font. Note that you can resize the Property Sheet and a splitter divides the property descriptions from the values, so you can choose a font large enough to read and still see complete text.

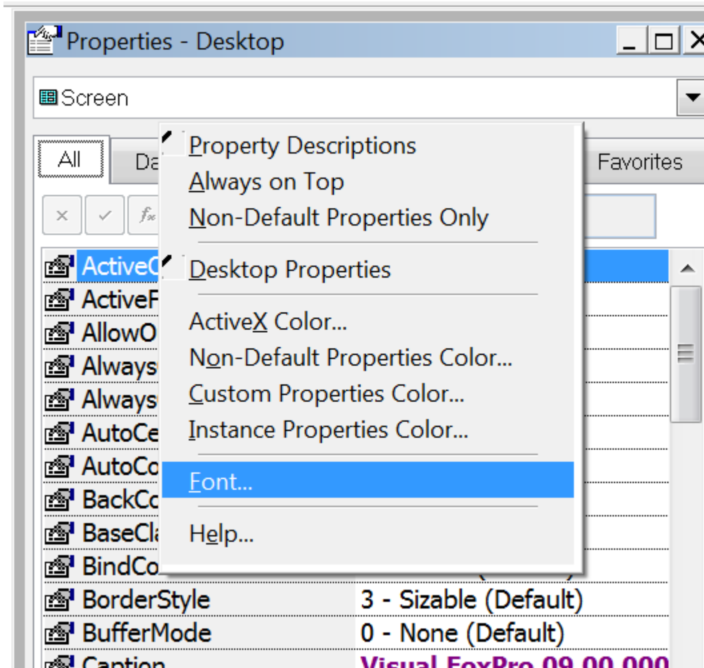

Figure 3. To change the font of the Property Sheet, you have to know where to right-click.

Set up colors

Just as you can choose fonts and sizes, VFP lets you specify colors for various things beyond those controlled by Windows. As **Figure 3** indicates, there are quite a few color settings for the Property Sheet. **Table 2** shows what each choice affects; they're applied in the order shown in the table. For example, all PEMs added at the current level of the class hierarchy use the Instance properties color, whether they're changed or not.

Table 2. The Property Sheet now lets you use up to five different colors.

Color group	Includes
Instance Properties Color	PEMs added at the current level of the class hierarchy.
Non-Default Properties Color	PEMs set at the current level of the class hierarchy.
Custom Properties Color	PEMs added at any level other than the current level of the class hierarchy.
ActiveX Color	PEMs of any ActiveX controls.
Default	All other PEMs, that is, built-in PEMs not set at this level of the class hierarchy. Always black.

You can set the background and foreground colors of the VFP Desktop programmatically by specifying _SCREEN.BackColor and _SCREEN.ForeColor. **Figure 4** shows part of the desktop after issuing these commands from the Command Window (as well as setting _screen.fontsize, and issuing ?version())

```
_screen.BackColor = 16502457
_screen.ForeColor = RGB(168,52,255)
```

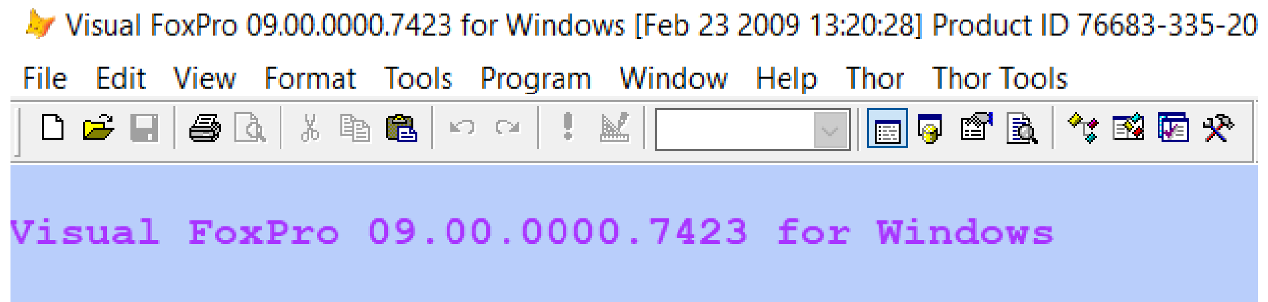

Figure 4. You can change the background and foreground color of the VFP desktop with a little code.

Until you've closed VFP with the new background color, you can reset it using MODIFY WINDOW Screen. While that command doesn't reset the ForeColor property, it resets what you see when you send output to the screen. It's probably best to simply reset all the properties you want to explicitly.

For code editing windows, the choices are different. You can specify a variety of settings for the foreground and background colors through the Syntax Color settings on the Editor tab of the Options dialog, shown in **Figure 5**. The Area dropdown lets you specify different font styles and colors for different elements of the code: see **Figure 6**.

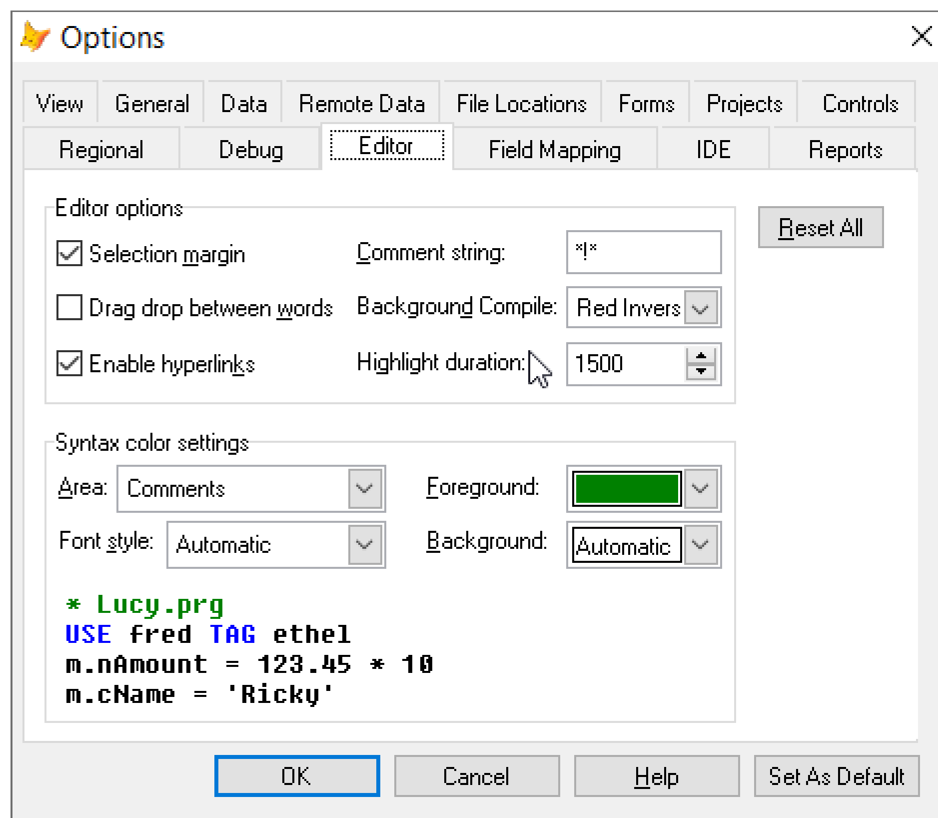

Figure 5. The Editor tab of the Options dialog lets you specify VFP's syntax coloring options.

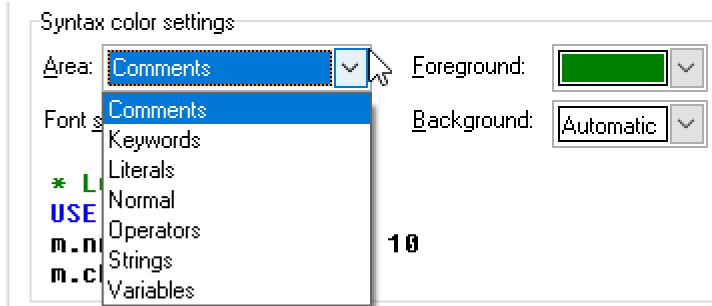

Figure 6. Use the Area dropdown on the Editor tab to choose different elements of the code for which to set color and style.

The choices here apply to all code windows, including those opened with MODIFY COMMAND, method editors, the stored procedures editor, and code windows opened by the Menu Designer. The sample code block at the bottom of the Syntax color settings section shows you the result of your choices.

In addition to syntax coloring, this tab controls what VFP calls "background compile." This feature allows you to have code marked as you type to indicate that it's not yet syntactically valid. For example, in **Figure 7**, the letters "rep" are red because so far, they're not valid. As soon as I type the "l", the string goes pink, as in **Figure 8**, to indicate that it's valid. When the line is complete, your specified syntax coloring goes into effect.

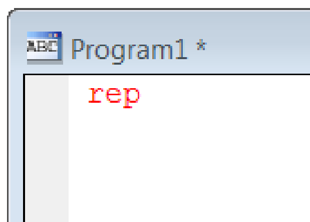

Figure 7. Background compile highlights code that's not yet valid. Once it becomes valid, the color or highlight changes.

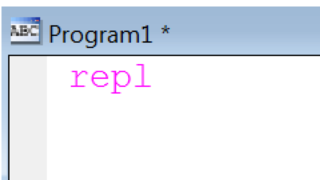

Figure 8. As soon as what you've typed is recognizable as VFP, the color changes.

In fact, there are several different settings for background compile and each behaves slightly differently. The example shown here uses the "red inversion" setting (my preference). The default "underline" setting underlines what you type until it becomes syntactically valid. As you complete a keyword or other element, it changes to the appropriate color, just as it would with background compile turned off. The final choice, "gray," keeps the whole line gray until it's valid, at which point syntax coloring takes effect. Be aware that you must have syntax coloring turned on to use background compilation.

Copyright 2022, Tamar E. Granor

Unlike your font choices, changes you make in the Editor tab apply only to the current VFP session unless you click Set as Default.

Set up shortcuts for common items

VFP offers several ways to set up quick ways to get things done. I'll highlight a few here, but this is not an exhaustive list.

Easy shortcuts with IntelliSense

IntelliSense provides one easy way to quickly type things. It's already configured to expand keywords once you've typed enough to identify them. It also comes with some built-in shortcuts. For example, MC is pre-configured as a shortcut for MODIFY COMMAND.

It's easy to add shortcuts for other commands you use often. For example, I've defined BL as BROWSE LAST because I use it so much. To define a simple shortcut like this:

- Open the IntelliSense Manager (Tools | IntelliSense Manager)
- Click the Custom tab
- In the Replace textbox, type the shortcut
- In the With textbox, type the expanded version
- Click Add

Figure 9 shows the set-up for my shortcut for BROWSE LAST, just before clicking Add.

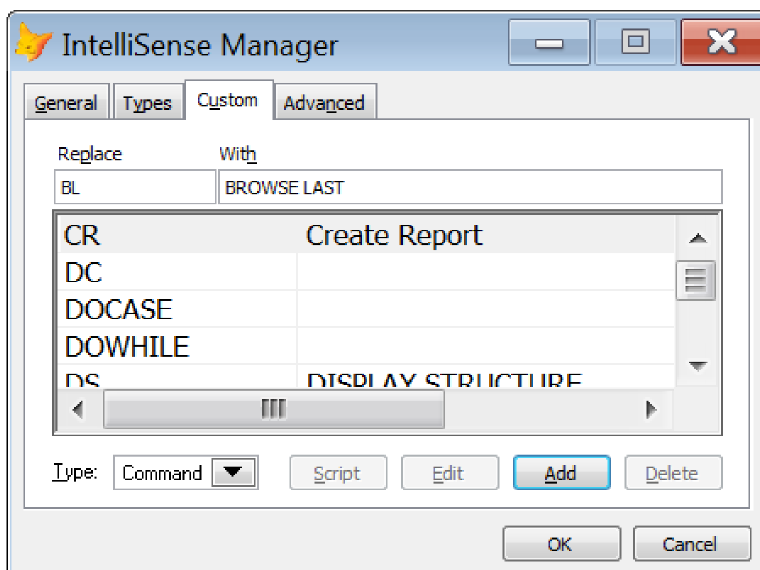

Figure 9. Adding shortcuts to IntelliSense is easy.

It's possible to add more complex items to IntelliSense as well, but they go beyond the scope of "Tips and Tricks"; with his permission, the paper for Rick Schummer's session "Inspiring Everyday Practical IntelliSense" is included with the downloads for this session. (You may also want to watch Jim Nelson's session on the VFPX IntelliSenseX project, which

extends VFP's native IntelliSense. A video is available at https://www.youtube.com/watch?v=F_ikTQ0kRFI.)

Save shortcuts in the Toolbox

The Toolbox provides another way to set up shortcuts. If you have a block of code or a comment you need often, put it into the Toolbox's Text Scraps section and drag and drop it when you need it. Think of this section as a clipboard capable of handling many clippings.

For example, I was cleaning up some code that didn't properly save and restore the work area in each routine. I needed to add this code at the beginning of each routine:

```
LOCAL nOldSelect
nOldSelect = SELECT()
```

and this code at the end of each:

```
SELECT (m.nOldSelect)
```

If there had only been a single block to add, I could have just put it on the clipboard and pasted it everywhere I needed it, but I had two blocks to paste. So I put them both into Text Scraps and dropped them where they were needed.

Adding a text scrap is easy. With the Toolbox open and the Text Scraps category (or whichever category you want to use) expanded, highlight the code you want to turn into a scrap and drag and drop it into the Toolbox, as in **Figure 10**. The item is added to the expanded category with a name of "Text:" plus the first line of the code block; see **Figure 11**. You can change the name of the item by right-clicking on it and choosing Rename.

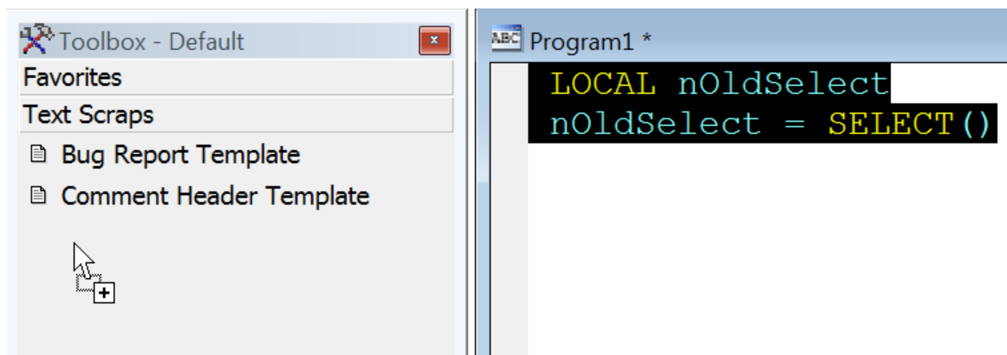

Figure 10. You can drop blocks of text into the Toolbox and then drag them back into code windows.

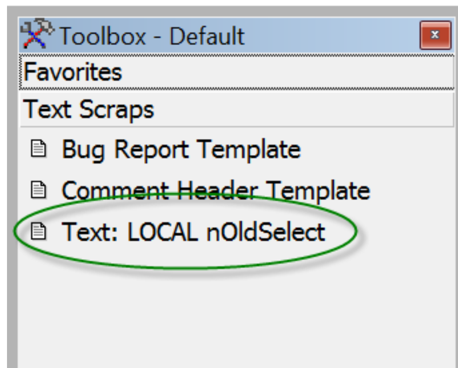

Figure 11. Code blocks go into the Toolbox with a default name of "Text:" plus the first line of code.

Once the code block is in the Toolbox, using it is simple. Just drag the item from the Toolbox and drop it where you want it in a code window.

Have a standard change comment

Whether you set it up with IntelliSense or the Toolbox, one important shortcut to set up is one that allows you to add a comment every time you change some code. The big advantage of doing it the same way each time is that it makes it easy to find changes. I like my change comment to include my initials and the date, which allows me to search for either of those items. I use an IntelliSense script (modified from one someone else shared a long time) for this. It's triggered by the string TEGMOD; **Listing 32** shows my script. When I use it, I get something like **Listing 2**; credit goes to Toni Feltman for suggesting prompting for the change description itself, so it doesn't get skipped.

Listing 1. This IntelliSense script inserts a standard change comment into my code. Modify it to set up your own standard change comment.

```
lparameters toFoxCode
local lcReturn

if toFoxCode.Location <> 0
   toFoxCode.ValueType = 'V'
   lcReturn = GetText()
endif toFoxCode.Location <> 0
return lcReturn

function GetText
local lcText, nDay, cMonthName, nYear, dToday
dToday = date()
nDay = day(dToday)
cMonthName = cmonth(dToday)
nYear = year(dToday)
cComment = inputbox("Reason for change")
text to lcText textmerge noshow
* Modified <<nDay>>-<<cMonthName>>-<<nYear>> by TEG
* <<cComment>>~
endtext
return lcText
```

Listing 2. My IntelliSense shortcut for a standard change comment generates this, where the second line contains whatever I type in when prompted.

```
* Modified 21-July-2014 by TEG
* Added new parameter to indicate direction.
```

I also have an IntelliSense script for a "TO DO" comment, and another comment script that's customized to the format one of my clients prefers, which is different from my standard comment.

Use keystrokes to speed things up

Beyond IntelliSense shortcuts (whether built-in or custom), there are lots of other places in VFP where a key combination can save you considerable time.

One that's been in VFP for many, many versions, but many people still don't know about lets you clear away all the windows temporarily so you can see the desktop. Just hold down Ctrl+Alt+Shift; as long as you do, the desktop is revealed. When you release the keys, everything is right back where it was. Very handy when you've, for example, issued DISLAY STRUCTURE (or DS, using my custom IntelliSense shortcut) to get a look at a table's list of fields.

I can't believe how long it took me to discover this next one. When you're working in the Method Editor (that is, the window that lets you edit method code in the Form or Class Designer), you can quickly navigate the dropdowns for object and procedure by starting to type the name you want. For example, in **Figure 12**, the Method Editor is currently showing the Init method, but when I open the dropdown and type "st", "StartPolling" is highlighted. I just have to hit Enter to go to that method.

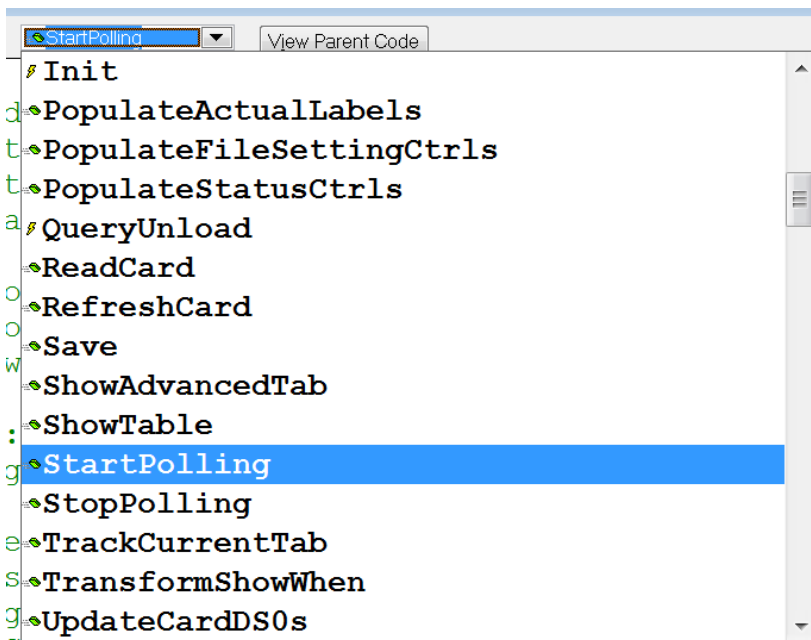

Figure 12. Start typing the name of the method you want to edit to get there quickly.

It's hard to know how to classify this next one. GetFile() and the other similar functions remember the folder you last selected. When you use them a second time, they start out pointing to that folder. If you want to reset them to point to the folder where you're currently working, use the CD command and point to the current folder. That folder is almost always the first item in the MRU list that appears when I type CD, but if not, CD CurDir() does the trick.

For example, if I'm working in the folder containing examples for this paper and call GetFile(), I start out in that folder. If I then navigate to the VFP home folder and select a file from there, the next time I call GetFile() (or PutFile() or LocFile() or GetDir()), it starts out in the VFP home folder. But if I execute CD CurDir() in the Command Window before calling the function, it again starts in the folder containing this session's examples.

This one isn't a keystroke, but it feels similar. When a Browse window including a memo field is open, you can hover over the memo field to see what it contains, as in **Figure 13**.

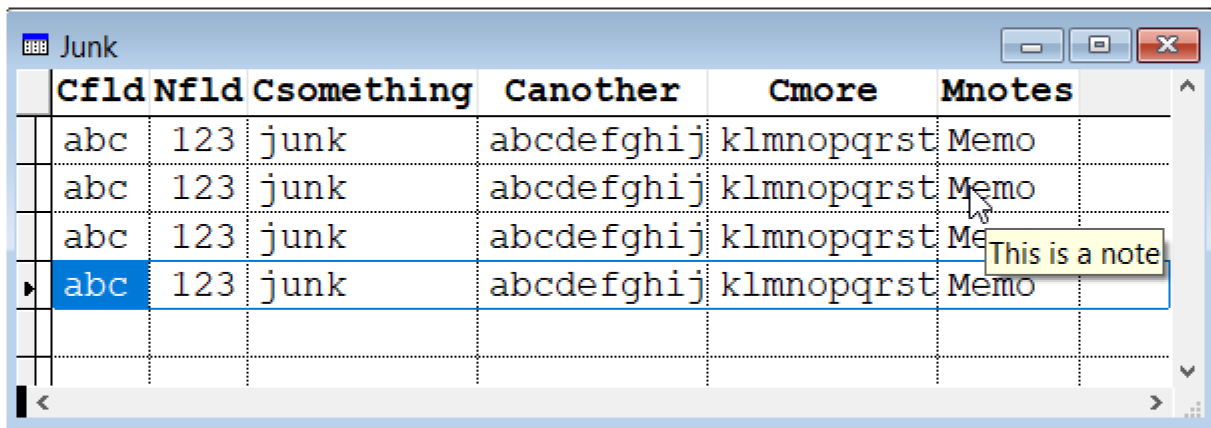

Figure 13. When you hover over a memo field in a Browse window, the memo contents appear in a tooltip.

Let Property Editors make things easier

VFP 9 gave us another way to customize the Property Sheet. Property Editors are like tiny Builders. They let you change the way the value is specified, as well as letting you run code after the user specifies a value.

Some built-in properties use specialized editors. For example, if you double-click on the various color properties (BackColor, ForeColor, SelectedBackColor, etc.) or click on one of those properties and then click the ellipsis button in the Property Sheet (highlighted in red in **Figure 14**), the Color Picker opens. Double-clicking a logical property in the Property Sheet toggles its value. For properties with a fixed list of values (like Alignment), double-clicking goes to the next value. That's all available without any code, but only for built-in properties.

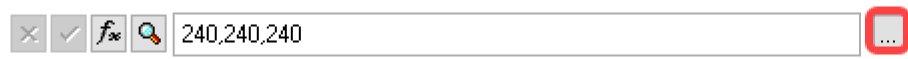

Figure 14. Use the ellipsis button in the Property Sheet to access a property editor. Double-clicking the property or its value does the same thing.

Property editors let us do things like this for our custom properties, but they also let us do much more. For example, in one application, I use property editors to adjust the position of controls in a container class after setting a caption for one of them. In the same application, I use a property editor to resize labels after changing their captions, even if AutoSize is off for those labels.

To create a property editor, choose MemberData Editor from the Form or Class menu. (If you install Thor's PEM Editor, you'll have the MemberData Editor available from the context menu in the Property Sheet.) The MemberData editor opens, as in **Figure 15**. To create or edit a property editor, click the magnifying glass next to Script.

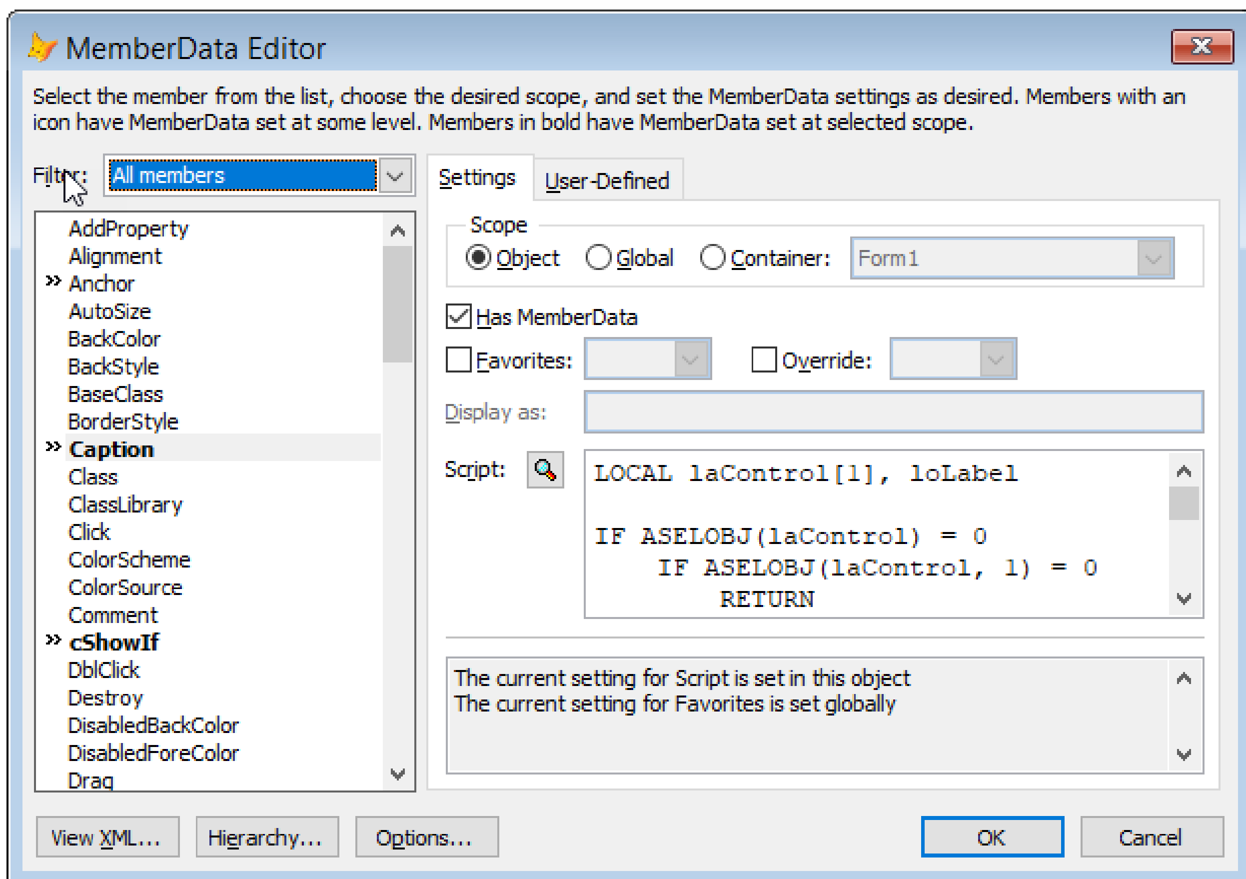

Figure 15. The MemberData Editor lets you set up property editors as well as determine how a property is capitalized and more.

An editing window opens, empty for a new property editor or showing the code of an existing property editor. **Listing 3** shows my property editor for the Caption property of labels. It prompts for the caption text and then uses a trick to resize the label to the exact width needed for the new caption. (Why don't I just set AutoSize on in my "base" label class? Because many of the labels in this application need to be right-aligned, and AutoSize doesn't play well with right alignment.)

Listing 3. This property editor is used to modify the caption for a label and make it exactly the right size at the same time.

```
LOCAL laControl[1], loLabel

IF ASELOBJ(laControl) = 0
   IF ASELOBJ(laControl, 1) = 0
      RETURN
   ENDIF
ENDIF

loLabel = laControl[1]

LOCAL lcCaption

lcCaption = INPUTBOX("Label caption","Specify label caption")
IF NOT EMPTY(m.lcCaption)
   loLabel.Caption = m.lcCaption
   loLabel.AutoSize = .T.
   loLabel.Width = loLabel.Width
   loLabel.ResetToDefault("AutoSize")
endif

RETURN
```

The first block of code is similar to what you use in a Builder. It uses ASELOBJ() to get an object reference to the relevant object. If that's successful (and in this situation, it always will be because you can't run this property editor unless a label is selected in the Form or Class Designer), it uses INPUTBOX() to prompt for a caption. If a new caption is specified, the Caption property of the label is set. Then, AutoSize is set to .T. to automatically get the appropriate width. To keep that width even after resetting AutoSize, the Width property is assigned to itself. Finally, AutoSize is reset to its default value.

As the previous example shows, a property editor can actually change more than one property. The same application uses a custom container control rather than checkboxes. The container holds a label and a small textbox, and the user can change it from Y to N with the keyboard or by double-clicking. An instance of this class is shown in **Figure 16**.

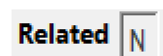

Figure 16. This container class is used instead of checkboxes in an application. The user can type Y or N or double-click to toggle the value.

The container class has a custom property named cLabelCaption to specify the caption of the contained label. That property has a property editor (shown in **Listing 4**) that prompts for the label, stores it in cLabelCaption and in the Caption property of the contained label, and then resizes the label, repositions the textbox, and resizes the whole container.

Listing 4. This property editor collects the desired caption, and then adjusts size and position in the container.

```
LOCAL laControl[1], loContainer, lcLabel, loLabel, loTextbox

IF ASELOBJ(laControl) = 0
   IF ASELOBJ(laControl, 1) = 0
      RETURN
   ENDIF
ENDIF

loContainer = laControl[1]
loLabel = loContainer.lblYN
loTextbox = loContainer.txtYN
lcLabel = INPUTBOX("Enter label for this YN control")
IF NOT EMPTY(m.lcLabel)
   loContainer.cLabelCaption = m.lcLabel
   loLabel.Caption = m.lcLabel

   * Now fix size and width
   loLabel.AutoSize = .T.
   loLabel.Width = loLabel.Width
   loLabel.ResetToDefault("AutoSize")

   loTextbox.Left = loLabel.Width + 5
   loContainer.Width = loTextbox.Left + loTextbox.Width
ENDIF
RETURN
```

This property editor and similar ones used for other container controls in this application save us a tremendous amount of time when designing forms. We don't have to stop and carefully move things around. They just work, at both design-time and run-time.

This is a small sample of what you can do with property editors. For more, see my article at https://tinyurl.com/474k74aw, and/or Doug Hennig's introduction to _MemberData at https://doughennig.com/Papers/Pub/200406dhen.pdf. Doug also set up a framework for property editors; read about it at https://tinyurl.com/f7n96xb3.

See your settings

Sometimes, you need to see what settings you've specified in the Tools | Options dialog. To do so, open the Debugger, making sure the Debug Output window is open. Then, hold the Shift button down while clicking OK and the list of settings appears in the Debug Output window. Those that can be written as VFP code are shown that way. The others, which correspond to registry settings, are shown as comments, as in **Listing 5**, which shows partial results.

Listing 5. Holding down the Shift key while clicking OK in the Options dialog echoes the whole list of settings to the Debug Output window.

```
SET TALK ON
SET NOTIFY OFF
```

```
SET CLOCK OFF
&& RecentlyUsedFiles = 6
&& DisplayCount = 15
SET COMPATIBLE OFF
SET PALETTE ON
SET BELL OFF
SET SAFETY ON
SET ESCAPE OFF
SET LOGERRORS ON
SET KEYCOMP TO WINDOWS
SET CARRY OFF
```

Move your settings between machines

If you're changing from one machine to another, you can grab the list of settings described in the previous section and run it on the new machine. (Right-click in the Debug Output window and choose Save As to save it to a file.) That will give you all the settings that can be set by VFP commands. Once you've done that, open the Tools | Options dialog and click Set As Default.

To move the other settings from one machine to another, open the Registry Editor on the machine that's configured as you want. Expand the HKEY_CURRENT_USER hive and navigate to Software\Microsoft\VisualFoxPro\9.0. Right-click on that node and choose Export. Use the dialog that opens to save the settings from that branch into an REG file, as in **Figure 1**. Copy the file to the new machine and double-click it to add those settings to the Registry.

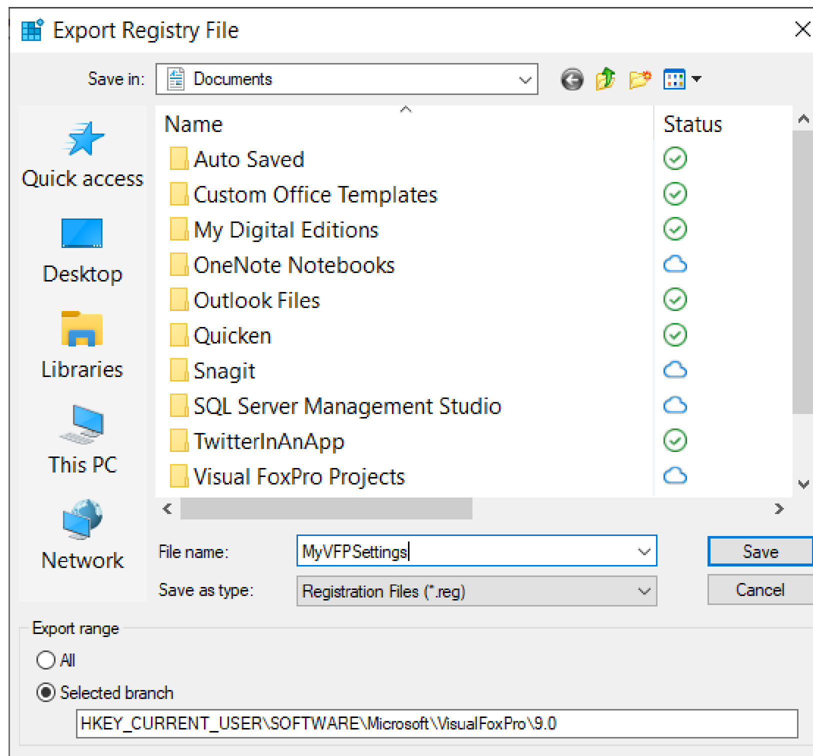

Figure 17. This dialog lets you save the registry settings for VFP to a file, which you can run on a new machine.

Debug more efficiently

Learning to use VFP's debugger well is beyond the scope of a "Tips and Tricks" session. (In fact, I've done a whole session on the topic. You'll find the white paper at https://tinyurl.com/5484j548.) But even without the whole lesson, there are a few little things you can do to make yourself more efficient with the debugger.

Watch window tricks

The Watch window lets you see the value of the expressions you specify. Keeping a few things there all the time speeds up debugging. First, put the MESSAGE() function in, so that after an error occurs and you suspend the code, you can still see what error message got you there. Of course, it's only as good as the last error that occurred, but most of the time, that's the one you're working on. Be aware, though, that the function can get confused. If the error is one of those that includes the name of something (like error 12, 'Variable "variable" is not found.'), as you work, the name may disappear and be replaced by something else in the Watch window, so do note right away what name is included.

Keep LineNo() in the Watch window because it allows you to set a breakpoint anywhere. If you just want code to stop, double-click the margin next to LineNo() and execution will stop on the next executable line. Very handy for cases where determining a specific breakpoint is tricky.

Another handy way to set breakpoints is to keep an expression like “INIT” $ PROGRAM() in the Watch window. When I want execution to stop at a particular method, I edit the first part of the expression, substituting the name of the method of interest and double-click to set a breakpoint. This can be much faster than opening a class or form, finding the right method and adding a breakpoint. It also gives you a chance to set a breakpoint generically, like “stop on any method named ‘Calculate’.” On the flip side, if you want to stop at a specific control’s method, put the end of the name of the control in front of the method name. For example, if I have a control named txtAmount and I want to stop when its Valid method runs, I can make the expression “AMOUNT.VALID” $ PROGRAM().

It’s a little tricky to put a reference to a property of the current form or one of its controls into the Watch window. You can’t use THISFORM to refer the form because THISFORM isn’t always meaningful. Instead, use _SCREEN.ActiveForm to refer to the form. For example, if I want to keep an eye on the Enabled property of a textbox called txtCustomerCode, I’d put this string in the Watch window:

```
_SCREEN.ActiveForm.txtCustomerCode.Enabled
```

Then, I can set a breakpoint when that property changes, or just check it periodically as I test the form. One warning: if you have a breakpoint on an expression using _SCREEN.ActiveForm, any time the form calls outside code (like a library routine), your breakpoint will fire because the expression can no longer be evaluated.

Use Find in the Debugger

Find (including the CTRL+F shortcut) works in all of the Debugger’s windows except Call Stack. That means you can easily find a particular variable, watch expression, line of code, or string you’ve sent to Debug Output while debugging. Note that in the Trace window, Find works only for the displayed code. You can’t use it to find something in another method of the same form or class, as you can in the development environment.

Use DEBUGOUT to show you what’s happening

The Debug Output window gives you a way to track selected information while debugging without leaving traces a user can see. Instead of using WAIT WINDOW or MESSAGEBOX() to display debugging messages, use the DEBUGOUT command. As long as the Debugger is open, the messages are sent to the Debug Output window, but with the Debugger closed and at runtime, the messages don’t appear.

DEBUGOUT is very friendly, too. It accepts a string of arguments, and you don’t have to convert them to character. So, you can issue a command like:

```
DEBUGOUT "Pass: ", m.nPass, "started at ", m.tStart
```

to get results like **Figure 18**.

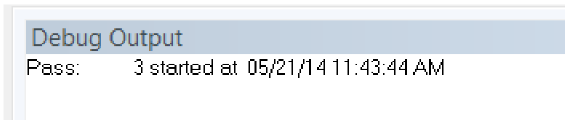

Figure 18. DEBUGOUT makes it easy to track what's going on while debugging. You can mix and match data of all types in a single call without converting to character.

Report Designer

Like the Debugger, the Report Designer is much too big a topic to cover in a Tips and Tricks session, but there are a few hints that can make you much more productive. (Check out Cathy Pountney's writings for in-depth coverage. She wrote a whole book about the Report Designer; it's available as an e-book from Hentzenwerke Publishing. After the major updates to reporting in VFP 9, two of her papers were published by Microsoft. You'll find them at https://tinyurl.com/5dn8fbzj and https://tinyurl.com/2p8rybcx.)

Set up reporting defaults

Like other aspects of VFP, you can set defaults for the Report Designer in the Options dialog. This includes the default font and size and whether reports use a private data session by default. Perhaps most importantly, you can ensure that information about specific printers isn't stored in reports (**Figure 19**). As in other cases, make sure to click Set As Default to have your choices apply to more than the current VFP session.

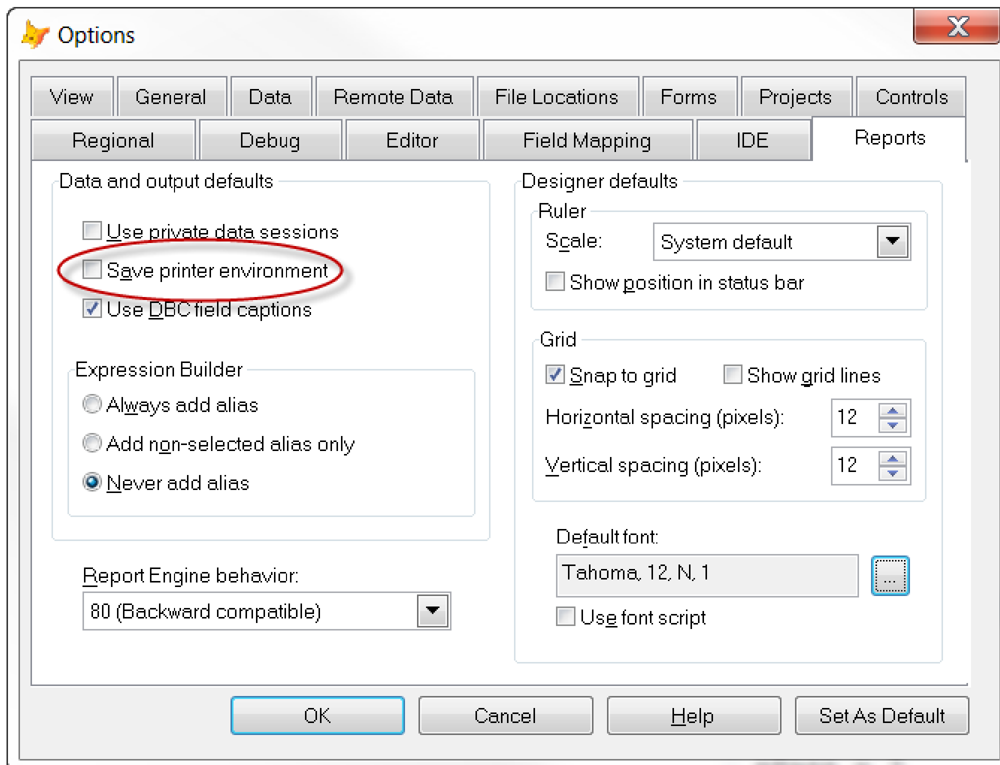

Figure 19. The Reports page of the Options dialog lets you set things up so that you don't have to spend time configuring each time you create a new report.

Use conditional formatting

Service Pack 2 (SP2) for VFP 9 introduced several new features in the Report Designer. Many of them are fairly complex, but one provides a simple solution to a long-standing problem: how to show data differently based on various conditions.

The issue is how you can, for example, show negative values in red, or bold certain data. The traditional solution to this problem is to put two fields on top of each other and set the Print When condition for each so that it only one shows at a time. VFP 9 introduced another solution, using a report listener. (You can find my write-up of both of those solutions at http://tinyurl.com/mvfz4ua.) But both approaches are still harder than the problem warrants.

SP2 for VFP 9 introduced the Dynamics tab that lets you set up conditions and corresponding formatting. For each field in the report, you can specify one or more conditions and associate specific formatting with that condition.

Consider a very basic inventory report for the Northwind database that just shows a list of products and the number of units on hand for each. One way to make the report more useful would be to highlight in some way (such as showing in red) items that need to be reordered; **Figure 20** shows part of such a report.

Inventory

Name	In stock
Alice Mutton	0
Aniseed Syrup	13
Boston Crab Meat	123
Camembert Pierrot	19
Carnarvon Tigers	42
Chai	39
Chang	17
Chartreuse verte	69
Chef Anton's Cajun Seasoning	53
Chef Anton's Gumbo Mix	0
Chocolade	15
Côte de Blaye	17

Figure 20. The Dynamics tab, added to the Report Designer in VFP SP2, makes it easy to do formatting-on-the-fly.

To set up such formatting, open the Properties dialog for the UnitsInStock field, then click the Dynamics tab and click Add. You're prompted to give the condition a name. Once you do so, the Configure Dynamic Properties dialog (**Figure 21**) appears. In the Apply when this condition is true textbox, put the condition to be tested; you can click the ellipsis button to use the Expression Builder. The rest of the dialog lets you specify the formatting. Note that

you can even replace the field with a different expression, so you might replace an empty value with a string like "None."

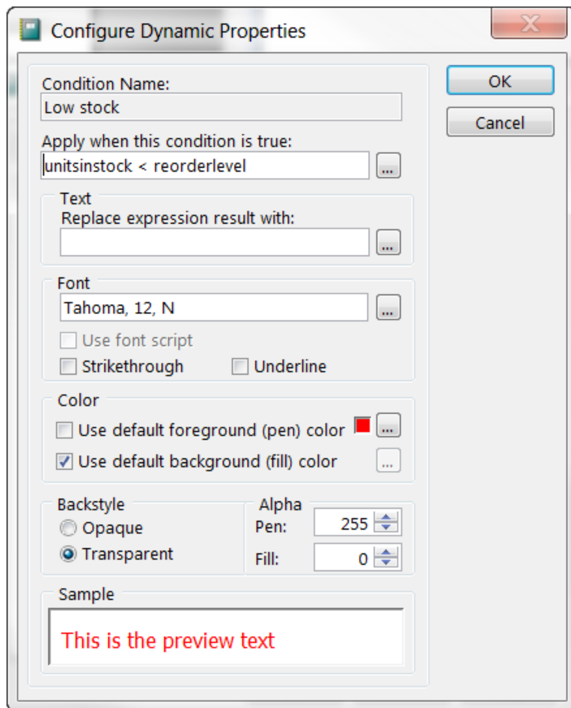

Figure 21. In this dialog, you specify the condition under which this formatting applies, and the formatting itself.

You can specify multiple conditions for a single field (which is presumably why you need to give each a name). Except for the "<default>" condition shown at the top, they're applied in the order in which they're shown in the Dynamics tab (**Figure 22**). Think of the list as a CASE statement. (In fact, it really is. Click the Script button to see the code generated from your conditions.)

Copyright 2022, Tamar E. Granor

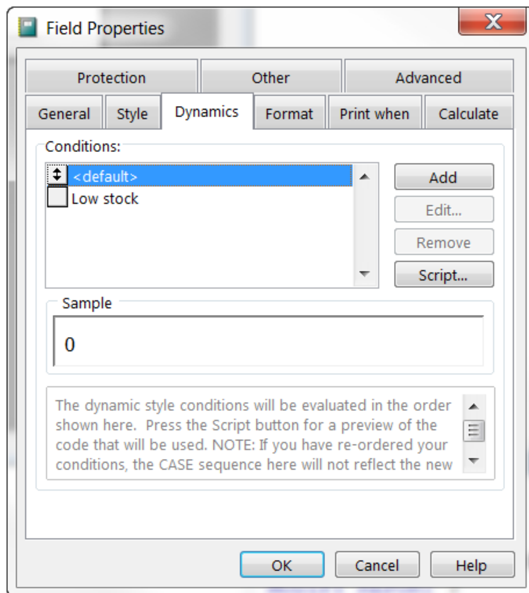

Figure 22. The Dynamics tab of the Field Properties dialog shows the conditions you've set up for the field.

You must SET REPORTBEHAVIOR 90 for the items in the Dynamics tab to take effect. The materials for this session includes the sample report, Inventory.FRX; you'll need to point it to the sample Northwind database.

Check out other tools

The tips and tricks above are about working with the tools that come with VFP. One of the best ways to become more productive is to take advantage of tools built by others. VFPX is a community project to add tools and components to VFP. You'll find it at https://vfpx.github.io/. It includes a wide range of projects that can make you more efficient.

In particular, Thor is both a tool and a manager of tools. I use some of the Thor tools literally dozens of times a day. I wrote about some of my favorites many years ago: https://tinyurl.com/mr237afu. (If I wrote that paper today, it would include some different things.)

Put the language to work for you

It's been said that if you can do something in Visual FoxPro, there are three ways to do it. While that's not really true, in many cases, there are multiple ways to get things done with one better than the others, or with different ones better in different cases. In this section, we'll look at some of those choices.

In addition, the VFP language has some features that are not immediately apparent and are worth looking at, as well as some that were added late in the game and may have escaped your notice. We'll examine some of those as well.

Working with strings

VFP has all kinds of facilities for working with character data. (In fact, there's so much there that several people have done whole conference sessions on it in the past. You'll find Rick Borup's paper "String Theory: Working with Serial Data in VFP" at https://tinyurl.com/yckj4hyw. Steve Black's paper "Text and String Handling in VFP" is at https://tinyurl.com/tvdhu5v.) Here are a few tricks for strings.

TRIM() functions handle more than spaces

In VFP 9, the string trimming functions (TRIM(), ALLTRIM(), LTRIM(), RTRIM()) were extended to accept one or more characters to be trimmed. Previously, these functions could remove only spaces at either end of a string. Now they remove all specified characters. For example, you can remove spaces plus CR and LF from both ends of a string with:

```
cMyString = ALLTRIM(m.cMyString, ' ', CHR(13), CHR(10))
```

Note that when you specify any characters to be removed, you have to explicitly specify space (CHR(32)) if you want it removed as well, as in the example. Note also that you can specify strings of more than one character to remove only the combined string.

Since you can include letters of the alphabet in the list of items to trim, there's also an optional parameter to indicate whether the trim is case-sensitive. By default, it is; pass 1 as the second parameter for a case-insensitive trim.

One handy place to use the new capability is when you've building a comma-separated (or anything else separated) list and you need to remove the extra separator at the end. For example, **Listing 6** (TrimTrailingComma.PRG in the materials for this session) shows code that builds a comma-separated list of customer countries. Rather than using a complex expression involving SUBSTR() and LEN() to remove the final separator, TRIM() does the trick. This example demonstrates the use of a multi-character string as the item to be trimmed.

Listing 6. TRIM's new ability to remove any specified characters simplifies the code to build a comma-separated list of items.

```
SELECT DISTINCT Country ;
   FROM Customers ;
   ORDER BY Country ;
   INTO CURSOR csrCountries

LOCAL cAllCountries
cAllCountries = ''

SCAN
   cAllCountries = m.cAllCountries + ALLTRIM(csrCountries.Country) + ", "
ENDSCAN

cAllCountries = TRIM(m.cAllCountries, ', ')
```

Parse with ALINES()

Just about every project I work on includes at least one situation where I have to take a long string and break it up. Whether it's dividing a piece of text into lines or breaking a comma-separated list into its individual components, it seems I always need to parse strings. Prior to VFP 6, breaking text into lines and breaking up a comma-separated list seemed like distinct tasks, but with the addition of the ALINES() function, parsing turned into one simple task.

ALINES() takes a string and parses it into individual array elements. You specify what determines the end of one item and the beginning of the next; by default, it's CR and LF in any combination, but you can specify any separators you want.

I use ALINES() most often to break up a comma-separated list of items, as in **Listing 7**. You can use it to break the VFP path into its constituent parts by specifying both comma and semi-colon as separators. It's also great when you want to take the text from a file and break it up for processing. I use it that way in several developer tools, where I break code into its component lines.

Listing 7. ALINES() makes it easy to break a string up into its components.

```
LOCAL aItems[1], nItems, cList
cList = "Red,Yellow,Blue,Orange,Green,Purple"

nItems = ALINES(aItems, m.cList, ",")
```

I've even had cases where I used ALINES() to break something into lines, and then applied it to the lines themselves to break them into their constituent parts.

ALINES() is faster than other parsing code, and as the string to be parsed gets longer, its advantage increases. (Execution time for ALINES() increases linearly with the length of the string, while execution time for other approaches to parsing increases at a higher rate.)

Initialize arrays with ALINES()

While I was originally preparing this session, I was shown another, very cool use for ALINES(): initializing an array. Some languages let you assign multiple values to an array in a single line of code. VFP doesn't have that syntax.

But ALINES() gives you a way to do it, either as a one-liner or in two (probably more readable) lines. **Listing 8** shows both versions. (Incidentally, most of the time, I'd prefer to have a table or cursor with the list of states and abbreviations rather than an array. Of course, you could use APPEND FROM ARRAY to put the data in the array here into a cursor or table.)

Listing 8. You can use ALINES() to initialize an array, as long as you're trying to fill it with character strings. You can do it in two lines or just a single line.

```
* In the two-line version, you set up the string first, then parse it.
cArrayData = "AL,AK,AZ,AK,CA,CO,CT,DE,FL,GA,HI,ID,IL,IN,IA,KS,KY,LA,ME,MD," + ;
```

```
             "MA,MI,MN,MS,MO,MT,NE,NV,NH,NJ,NM,NY,NC,ND,OH,OK,OR,PA,RI,SC," + ;
             "SD,TN,TX,UT,VT,VI,WA,WV,WI,WY"
ALINES(aStateAbbrev, m.cArrayData, ",")

* The one-line version creates the string inline.
ALINES(aStateAbbrev, ;
       "AL,AK,AZ,AK,CA,CO,CT,DE,FL,GA,HI,ID,IL,IN,IA,KS,KY,LA,ME,MD," + ;
       "MA,MI,MN,MS,MO,MT,NE,NV,NH,NJ,NM,NY,NC,ND,OH,OK,OR,PA,RI,SC," + ;
       "SD,TN,TX,UT,VT,VI,WA,WV,WI,WY", ;
       ",")
```

Read and write files easily

VFP 6 also gave us the FileToStr() and StrToFile() functions that let us easily read and write files. Not only is it easier to create and read text files with these functions, but they also allow you to create or read other types of files in one step.

I use StrToFile() for logging with a function or method like **Listing 9**. This example is drawn from the Object Inspector; it accepts a string to add to the log and adds it with a timestamp in front. (The materials for this session contain a very basic logging class as Logger.PRG.)

Listing 9. StrToFile() is great for logging activity in an application.

```
PROCEDURE LogIt
LPARAMETERS cLogString, lStartNewLogFile

IF EMPTY(This.cLogFile)
   This.cLogFile = FORCEPATH("Inspector.Log", SYS(2023))
ENDIF

STRTOFILE(TTOC(DATETIME()) + ":" + m.cLogString + CHR(13) + CHR(10), ;
          This.cLogFile, not m.lStartNewLogFile)

RETURN
```

FileToStr() combined with ALINES() provides an easy way to read and parse a text file. In fact, I've occasionally used that combination to parse a log file like the ones created using StrToFile() and dump it into a cursor, as in **Listing 10**, included in the materials for this session as ParseLog.PRG. (Why not create a cursor or table in the first place? Because I'm usually creating the log file for non-developers to be able to read.)

Listing 10. FileToStr() plus ALines() gives you a way to read and parse a file.

```
LOCAL cLog, cFileContent, aFileLines[1], nLineCount, nLine

cLog = GETFILE("LOG","Log to parse:")

IF NOT EMPTY(m.cLog)
   cFileContent = FILETOSTR(m.cLog)

   nLineCount = ALINES(aFileLines, m.cFileContent)
```

```
   DIMENSION aFileLines[m.nLineCount, 1]

   CREATE CURSOR csrLines (mLine M)
   INSERT INTO csrLines FROM ARRAY m.aFileLines

ENDIF
```

Text-merge tips

One of VFP's cooler string-related features is text-merge, the feature that lets you combine fixed text with data to produce a single string. Added to FoxPro 2.0 to facilitate the screen and menu generator programs, text-merge got better and better over the years.

Originally, using text-merge required two SET commands (SET TEXTMERGE TO, SET TEXTMERGE ON) and then a series of lines using \ and \\ to emit the desired strings. The TEXT command combines all of that into a single command block. One of the most common uses for TEXT is to build SQL strings to pass to a server, as in **Listing 11**, drawn from a class that interacts with QuickBooks data.

Listing 11. Use the TEXT command to build multi-line strings that merge in data.

```
* Convert name to SQL format
cSQLName = This.FormatSQLString(m.cName)

TEXT TO m.cSQL NOSHOW TEXTMERGE

   SELECT COUNT(*) as nHowMany
     FROM <<ALLTRIM( m.cTableName)>>
     WHERE Name = <<m.cSQLName>>
ENDTEXT
```

In another application, I needed to call PuTTY (a program for handling Telnet and SSH) programmatically. I stored the path to PuTTY, the name of a saved PuTTY session with the necessary settings, and the user's ID in a configuration table. One method retrieves that information and puts it into properties of the object making the call. Then, the code in **Listing 12** creates the necessary call and stores it in a batch file, so it can be run. The TEXT command generates a result like **Listing 13**.

Listing 12. Here, the TEXT command merges configuration information into a command line that calls PuTTY.

```
TEXT TO m.cCommand TEXTMERGE NOSHOW

"<<ALLTRIM(This.cPuTTYPath)>>" -load <<ALLTRIM(This.cPuTTYSession)>> -l
<<ALLTRIM(This.cPuTTYUID)>>
ENDTEXT
cBATFile = FORCEPATH("RunPuTTY.BAT", SYS(2023))
STRTOFILE(m.cCommand, m.cBATFile, 0)
```

Listing 13. The code in Listing 12 generates a result like this line.

```
"C:\Program Files (x86)\PuTTY\putty.exe" -load TRCONN -l tgranor
```

Copyright 2022, Tamar E. Granor

If you already have a string containing the text that requires text-merge, the TextMerge() function, added in VFP 7, lets you handle the whole process in one step. In one client application, we used XML as a mechanism to transfer data between a VFP object and a C# object. We stored XML "skeletons" in a table; then, we retrieved the one we needed and used TextMerge() to fill it in. For example, one of the skeletons looked like **Listing 14**. When we needed to use it, we populated the variable cXMLNodeList, and then called the code in **Listing 15**.

Listing 14. To prepare certain data from transport to another application, this XML "skeleton" was merged using TextMerge().

```
<?xml version="1.0" encoding="utf-16"?>
<ArrayOfAgent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<<cXMLNodeList>>
</ArrayOfAgent>
```

Listing 15. This code merges the "skeleton" above with the contents of a variable to produce an XML string to pass to another application.

```
IF NOT EMPTY(m.cXMLNodeList)
   cXMLSkel = This.GetXMLSkeleton("nodelistarray_skel")
   cXML = TEXTMERGE(m.cXMLSkel)
ELSE
   cXML = ""
ENDIF
```

Date and time tricks

FoxPro has always made it easy to work with dates and starting in VFP 3, with datetimes. The built-in date math makes it easy to figure out how many days between two dates or what date is 45 days from today. Beyond simple date math, there are a few other handy things you can do with dates and datetimes.

In VFP 6, the DATE() and DATETIME() functions were extended to make it easy to get your hands on a particular date or datetime value without having to worry about the SET DATE setting. Pass the components of the date or datetime you want and the functions hand you back the right value of the right type. For example, DATE(2022, 9, 28) gives you back a date value for September 28, 2022, while DATETIME(2009, 1, 20, 12, 0, 0) returns a datetime value for noon on January 20, 2009.

Using DATE() and DATETIME() to turn components into a value avoids complicated expressions involving conversion from numbers to strings, CTOD() or CTOT(), and control of the SET DATE setting. Even better, when you need to build one date from another, you can combine these functions with the DAY(), MONTH() and YEAR() functions to get exactly what you need. For example, if you have a variable containing a date and you need to get the first day of the same year, you can use code like **Listing 16**. Similarly, you can use HOUR(), MINUTE() and SEC() to extract part of one datetime to use in building another.

Listing 16. You can combine the DATE() and DATETIME() functions with the functions that extract portions of a date or datetime to build another date or datetime.

```
* Assume dBirth contains a person's birthdate.
dFirstOfBirthYear = DATE(YEAR(m.dBirth), 1, 1)
```

The GoMonth() function provides easy ways to get all kinds of date values. It handles simple things like this date one year from now (GOMONTH(dDate, 12)) or this date three months ago (GOMONTH(dDate, -3)), but it also can be used for some more complex items. For example, to find the last day of the month containing a particular date, use code like **Listing 17**. The first parameter passed to GOMONTH() there finds the first day of the month of the specified date (the specified date minus its days is the last day of the preceding month, so add 1 for the first of the same month). With a second parameter of 1, GOMONTH() then finds the first day of the next month. Finally, subtract 1 to get the last day of the specified month.

Listing 17. To find the last day of the month containing a particular date, figure out the first day, then use GOMONTH() to find the first of the next month and subtract one.

```
dLastOfMonth = GOMONTH(m.dDate – DAY(m.dDate) + 1, 1) – 1
```

When trying to get a complicated date calculation right, it's often best to work in the Command Window, and be sure to test a few extreme cases. Does it work right in leap year? In non-leap years? What if the input date is Feb. 28? What about Feb. 29? What if you cross a year boundary? Similarly, for datetimes, be sure to test whether your expression is right when you pass midnight.

Better use of SQL

VFP's SQL sublanguage makes many data tasks easy, but there are a couple of traps you can fall into. Plus, VFP offers some ways to help you make queries more efficient.

How many records were affected?

It's not unusual to want to know how many records a SQL command affected. (In fact, that's such a common request that SQL Server Management Studio shows that information each time you run some code.) VFP has long included the _TALLY system variable to answer that question. After you run SELECT, _TALLY contains the number of records in the result. For UPDATE or DELETE, _TALLY tells you the number of records affected.

In fact, _TALLY is also affected by quite a few Xbase commands, including REPLACE, APPEND FROM, and COPY TO. (The complete list is in the _TALLY topic in VFP help.)

Tip #1 is to use _TALLY rather than checking RECCOUNT() or ALEN() or using COUNT to see how many records are in a query result.

However, tip #2 is to be careful about that. Because _TALLY is affected by so many things, be sure to check it right after the command you're interested in.

However, in some applications, that may not be sufficient. If you have timers running or have other code that can interrupt execution (bound events, for example), it's probably best to stay away from _TALLY. In one large, multi-layered application I worked on in which code from one layer can interrupt code from another, we found we had to get rid of _TALLY, because occasionally such an interrupt would result in errors. I still use _TALLY in simpler code, but I keep the risks in mind.

String comparisons, SQL style

Most VFP developers are aware of FoxPro's unusual string-matching habits and the role of SET EXACT in determining how strings are compared. Just in case, here's a quick refresher.

When SET EXACT is OFF, strings are compared only until the string on the right ends. If they match to that point, the comparison is considered true. That is, with EXACT OFF, 'Smithsonian' = 'Smith' is true, but 'Smith' = 'Smithsonian' is false.

When you SET EXACT ON, strings must match exactly (except for trailing blanks) for the comparison to be true. So, with EXACT ON, both 'Smithsonian' = 'Smith' and 'Smith' = 'Smithsonian' are false. However, 'Smith ' = 'Smith' and 'Smith' = 'Smith ' are both true, despite the extra spaces.

However, SET EXACT has no effect on SQL commands. Instead, string comparisons in SQL are controlled by SET ANSI. Just like SET EXACT, when SET ANSI is ON, strings must exactly match, except for trailing blanks.

ANSI OFF, however, is a little different than EXACT OFF. When ANSI is OFF, strings are compared to the end of the shorter string, whichever side of the comparison operator it's on. If they agree to that point, they're considered the same. So, in a SQL command, when ANSI is OFF, both 'Smithsonian' = 'Smith' and 'Smith' = 'Smithsonian' are true.

In both Xbase and SQL code, you can use the exactly equals operator ("==") to avoid having to deal with either SET EXACT or SET ANSI. Be aware, though, that in SQL code, "==" ignores trailing blanks; in Xbase, they're significant.

See how VFP optimizes SQL

VFP has a pair of functions that let you look under the hood to see how it's optimizing SQL commands. You can use them to figure out whether adding some index tags or tweaking your code can speed things up.

SYS(3054) controls "SQL ShowPlan," a feature that asks VFP to show the tags it's using to optimize a SQL command and the order in which it's doing things. You can ask for information on filtering only or on filtering and joins; I've never found a reason (other than demos) to ask for filtering only. You also can indicate whether to include the query itself in the output; I generally like to do so. To see filtering and join information with the original query, pass 12 to SYS(3054). Pass 0 to the function to turn SQL ShowPlan off.

By default, SYS(3054) simply displays its output on the active form. Since VFP 7, you can pass the name of a variable as a parameter and the output is stored in that variable, instead. However, each SQL command overwrites the information already in the variable.

VFP 9 introduced SYS(3092) to provide a way to send the output to a file. Pass the name of a file to SYS(3092) and subsequent applications of SYS(3054) send output to the named file. Pass the empty string to turn off logging.

The code in **Listing 18** puts all this together. Passing the name of a variable to SYS(3054) (even when specifying an output file with SYS(3092)) ensures that the output doesn't appear on screen.

Listing 18. Combine SYS(3054) and SYS(3092) to send information about how VFP optimizes SQL commands to a file you can examine later.

```
SYS(3092, "Optim.Log")
SYS(3054, 12, "cGrabOutput")

SELECT CustomerID, ;
       COUNT(DISTINCT OrderDate) AS DatesOrdered, ;
       COUNT(OrderDate) AS TotalOrders ;
   FROM Orders ;
   GROUP BY 1 ;
   INTO CURSOR csrHowManyOrders

SYS(3054, 0)
SYS(3092, "")
```

I've written a number of times about how to interpret the output from SYS(3054). This article digs pretty deeply into it: https://tinyurl.com/3zsmne5p. This article tells you how to speed things up, once you see what's not optimized: https://tinyurl.com/3kwhe4ee.

Tips for better program structure

There are several easy things you can do to make your code stronger, faster, and easier to maintain.

Make variables and parameters local

When local variables were introduced in VFP 3, private variables immediately become almost obsolete. (Public variables have always been a bad idea.) Local variables can be seen only in the routine in which they are declared, making it virtually impossible for one routine to step on another's variables. So, the easiest tip on this front is to declare all variables local unless you have an explicit reason for using another scope. (Thor includes a tool that will make the local declarations for you, which makes it easy to ensure that all variables you use are declared.)

Be aware that undeclared variables are private. That is, if you use a variable, but don't declare it, it's private, and thus can be seen in routines called by the current routine. Thus, the risk of failing to declare a variable local is that, in fact, it then becomes private and can

be seen by any routine called by the routine that created it. The easiest way to demonstrate this problem is with a recursive routine (that is, one that calls itself). The program in **Listing 19** (DrillDownFoldersLoopVarUndeclared.PRG in the materials for this session) recursively drills down the folder hierarchy to fill an array with the list of all subfolders of a specified folder. However, the loop variable nFolder is undeclared and thus private. Very quickly, this code runs into an infinite loop. Add a local declaration for nFolder and all is well; the materials for this session include DrillDownFolders.PRG to demonstrate. (In fact, even an explicit private declaration for nFolder solves the infinite loop problem since you get a new instance of the variable each time you call the routine.)

Listing 19. Failing to declare a variable can lead to confusing behavior. In this recursive case, it results in an infinite loop.

```
* DrillDownFolders.PRG
* Fill an array with the list of all
* folders and subfolders in a particular
* folder. Recursive.

LPARAMETERS aResults, cFolder, lStartOver
   * aResults = array to hold results.
   * cFolder = start folder
   * lStartOver = should we empty aResults first.

LOCAL aCurFolder[1], nFolders
LOCAL nResultCount, cOrigFolder, cFoundFolder

WAIT WINDOW "Processing folders inside " + m.cFolder NOWAIT

SET ESCAPE ON

IF m.lStartOver
   DIMENSION aResults[1]
   nResultCount = 0
ELSE
   nResultCount = ALEN(m.aResults, 1)
ENDIF

* Hold current folder
cOrigFolder = SYS(5) + CURDIR()

* Switch to specified folder
CD (m.cFolder)

* Get list of contained folders
nFolders = ADIR(aCurFolder, "", "D")

* Loop through list. We can start at position 3
* because 1 and 2 are always "." and ".."
FOR nFolder = 3 TO m.nFolders
   * First add it, adding the starting path
   nResultCount = m.nResultCount + 1
   DIMENSION aResults[m.nResultCount]
   cFoundFolder = ADDBS(m.cFolder) + aCurFolder[m.nFolder, 1]
```

```
   aResults[m.nResultCount] = m.cFoundFolder

   * Now, drill down
   DrillDownFoldersLoopVarUndeclared(@aResults, m.cFoundFolder, .F.)

   * Reset folder count
   nResultCount = ALEN(m.aResults, 1)
ENDFOR

CD (m.cOrigFolder)

RETURN m.nResultCount
```

On the other hand, declaring a variable private doesn't actually create it; it simply reserves that name as private in case you create such a variable. To create a private variable, you have to actually assign it a value. **Listing 20** demonstrates the trap this creates. The main program defines a private variable called cPrivate, but doesn't create it. When the procedure SubProc uses such a variable, although it's private, it's not the same variable.

Listing 20. Declaring a variable private doesn't actually create the variable. Private variables can only be created by giving them a value.

```
* Declaring a variable private doesn't create the
* variable, just reserves the name as private.

PRIVATE cPrivate

DO SubProc

* The next line generates an error
ON ERROR WAIT WINDOW MESSAGE()
?m.cPrivate
ON ERROR

RETURN

PROCEDURE SubProc

* The variable here is private (because it's not declared)
* but it's not the same variable as in the main program

cPrivate="abc"
?m.cPrivate

RETURN
```

Like other variables, all parameters should be local, for the same reasons. The PARAMETERS statement declares the listed items private. To make them local, use LPARAMETERS instead. Listing parameters in the routine's header also makes them local.

Count parameters correctly

The PARAMETERS() function that tells you how many parameters were passed into a routine goes way, way back, but it has a weird quirk. It always tells you the number of parameters passed to the last-called routine, which may not be the one you're in. The PCOUNT() function (which comes from dBase and was added in FoxPro 2.6) behaves the way you'd expect—it always returns the number of parameters passed to the routine in which it is called. **Listing 21** shows the difference; both functions are called right after entering the procedure, then another procedure is called, then the two functions are called again. **Listing 22** shows the results.

Listing 21. PARAMETERS() and PCOUNT() don't always return the same value. Use PCOUNT() for accuracy.

```
* Demonstrate PARAMETERS() vs. PCOUNT()

Subproc("abc", 123)

RETURN

PROCEDURE Subproc(cParm1, nParm2)

? "Immediately on entry to Subproc"
? "  PARAMETERS() returns ", PARAMETERS()
? "  PCOUNT() returns ", PCOUNT()

Subsubproc()

? "After call to Subsubproc"
? "  PARAMETERS() returns ", PARAMETERS()
? "  PCOUNT() returns ", PCOUNT()

RETURN

PROCEDURE Subsubproc

RETURN
```

Listing 22. The output from Listing 21 shows the difference between PARAMETERS() and PCOUNT(). PARAMETERS() is sensitive to other calls to user-defined routines.

```
Immediately on entry to Subproc
  PARAMETERS() returns    2
  PCOUNT() returns    2
After call to Subsubproc
  PARAMETERS() returns    0
  PCOUNT() returns    2
```

Use the right loop

VFP offers four different ways to write loops: DO WHILE, FOR, SCAN, and FOR EACH. Each of them is appropriate in different circumstances. To loop through a table or cursor, use SCAN. For counted loops, use FOR. To loop through collections, use FOR EACH. For anything else, use DO WHILE.

Copyright 2022, Tamar E. Granor

Why does it matter? First, some types of loops are faster than others. For example, a FOR loop is about an order of magnitude faster than the equivalent DO WHILE. A SCAN loop is almost always faster than DO WHILE NOT EOF(). FOR EACH is generally twice as fast as FOR.

In addition, the loops other than DO WHILE provide extra services. SCAN starts at the top of the table or cursor (unless you include the WHILE clause), goes to the next record automatically on each pass, and automatically reselects the work area you started in at the end of each pass. That's a lot of code you don't have to write each time.

Similarly, FOR automatically increments its counter on each pass. FOR has one behavior that you could see as a plus or a minus. It evaluates its endpoint only once, when you first enter the loop; even if you change a value used in the expression for the end value within the loop, the number of passes doesn't change.

Be aware of FOR EACH quirks

FOR EACH was designed to make it easy to loop through collections. The specified variable takes on the value of the next element in the collection on each pass. FOR EACH has a couple of behaviors that can be confusing.

First, it's possible for FOR EACH to miss some members of a collection. If the code in the loop removes items from the collection, you'll skip some items (which tells you something about how FOR EACH is implemented internally). In that case, you should use FOR with a negative increment.

When you use FOR EACH with VFP's native collections (which were added three versions after FOR EACH), you can get some weird behavior. Fortunately, VFP 9 introduced the FOXOBJECT keyword, which both resolves the weirdness and makes FOR EACH loops on native collections an order of magnitude faster that loops that don't use it. **Listing 23** shows one of the strange behaviors resolved by FOXOBJECT, that not all PEMs of the object are seen.

Listing 23. When looping through native collections, using FOXOBJECT avoids weird behavior.

```
LOCAL oColl, oItem, nMembers, aMemberList[1]

oColl = CREATEOBJECT("Collection")

oColl.Add(CREATEOBJECT("cusItem", "First", 1))
oColl.Add(CREATEOBJECT("cusItem", "Second", 2))
oColl.Add(CREATEOBJECT("cusItem", "Third", 3))

* Loop without FOXOBJECT
FOR EACH oItem IN m.oColl
   nMembers = AMEMBERS(aMemberList, m.oItem, 3)
   ? "Member count for item " + oItem.cName + " = " + TRANSFORM(m.nMembers)
ENDFOR

* Loop with FOXOBJECT
```

```
FOR EACH oItem IN m.oColl FOXOBJECT
   nMembers = AMEMBERS(aMemberList, m.oItem, 3)
   ? "Member count for item " + oItem.cName + " = " + TRANSFORM(m.nMembers)
ENDFOR

RETURN

DEFINE CLASS cusItem AS Custom

cName = ''
nOrder = 0

PROCEDURE Init(cName, nOrder)

This.cName = m.cName
This.nOrder = m.nOrder

RETURN

ENDDEFINE
```

OOP tricks

VFP has a rich set of object-oriented abilities, but some things about them may not be obvious. Here are a few tricks to make it easier to work with objects.

What's there?

Two functions, SYS(1270) and AMouseObj(), let you find out about the object under the mouse or, in the case of SYS(1270), at a specified location. These functions work both at design-time and at runtime. The syntax for these functions is shown in **Listing 24**.

Listing 24. The AMouseObj() and SYS(1270) functions both let you get a handle to an object at runtime or design-time.

```
nElements = AMouseObj( ArrayName [, nRelativeToForm])
uReturn = SYS(1270 [, nXCoord, nYCoord ] )
```

AMouseObj() provides more data than SYS(1270). It puts four items in the array: an object reference to the object under the mouse, an object reference to that object's container, and the column and row (in pixels) of the mouse position. If you pass the optional second parameter, the second, third and fourth elements are based on the outermost container. In that case, at runtime or when the mouse is over a form being designed, the second element of the array is an object reference to the containing form; if the mouse is over the Class Designer, the second element is an object reference to the class being designed. When that nRelativeToForm parameter is passed, the third and fourth elements of the array measure the mouse position relative to the object referenced in the second parameter.

I use AMouseObj() in code that works around a bug related to tooltips in grids in VFP SP2. The bug means that the grid-level tooltip is shown instead of the tooltip from the control inside the grid. I use AMouseObj() to find the object under the mouse and show its tooltip. The code, which goes in the grid class's ToolTipText_Access method, is shown in **Listing**

Copyright 2022, Tamar E. Granor

25. (The materials for this session include Base.VCX, which has a grid class that contains this code, and CustomerGrid.SCX, which demonstrates its use.)

Listing 25. This code goes in the grid's ToolTipText_Access method to work around the bug that shows only the grid's tooltip instead of the contained control's tooltip.

```
LOCAL cTip

* Let components have their own tooltips.
* Look up the tooltip for the object currently under the mouse.
LOCAL aMousePos[1], oColumn, oControl

cTip = ""

IF AMOUSEOBJ(aMousePos) > 0
   oColumn = aMousePos[1]
   IF NOT ISNULL(m.oColumn) AND UPPER(oColumn.BaseClass) = "COLUMN"
       * First, grab column-level tip in case we don't find something below
       cTip = oColumn.ToolTipText

       * Now, look for the right control.
       oControl = EVALUATE("oColumn." + oColumn.CurrentControl)
       IF NOT EMPTY(oControl.ToolTipText)
          cTip = oControl.ToolTipText
      ENDIF
   ENDIF
ENDIF

RETURN m.cTip
```

While SYS(1270) provides only an object reference, it has flexibility that AMouseObj() doesn't. You can pass a point (that is, X and Y coordinates) and the function tells you what's under the specified point. There are two tricky issues here. First, the coordinates you pass are relative to the screen, not to the form you're running or even to VFP. So, you may need to add _VFP.Left and _VFP.Top, respectively, to the points you're interested in to get the right answer. The second issue is that if you specify a point that isn't inside VFP, the function returns .F., so you need to check the return value's type before treating it as an object.

Many VFP developers set up a hotkey using SYS(1270) to let them get an object reference while testing code, as in **Listing 26**.

Listing 26. A hotkey using SYS(1270) gives you quick access to whatever's under the mouse.

```
ON KEY LABEL F11 o=SYS(1270)
```

Get IntelliSense for objects

VFP's IntelliSense makes writing accurate code much faster. But in some situations, IntelliSense doesn't automatically kick in. For example, when you're in a code window for a form or class method, while using the keyword THIS provides IntelliSense, references to

other objects do not provide a list of PEMs for the referenced object. In PRG windows, by default, there's no list of PEMs for any object reference.

Fortunately, there's an easy way to tell VFP to give you that list. Declare the relevant variable local and include the optional AS clause to tell VFP the variable's type. (You can add AS to a parameter declaration as well.) **Figure 23** demonstrates; oForm is declared as Form, so when I type oForm followed by a period, I see the list of PEMs for a form.

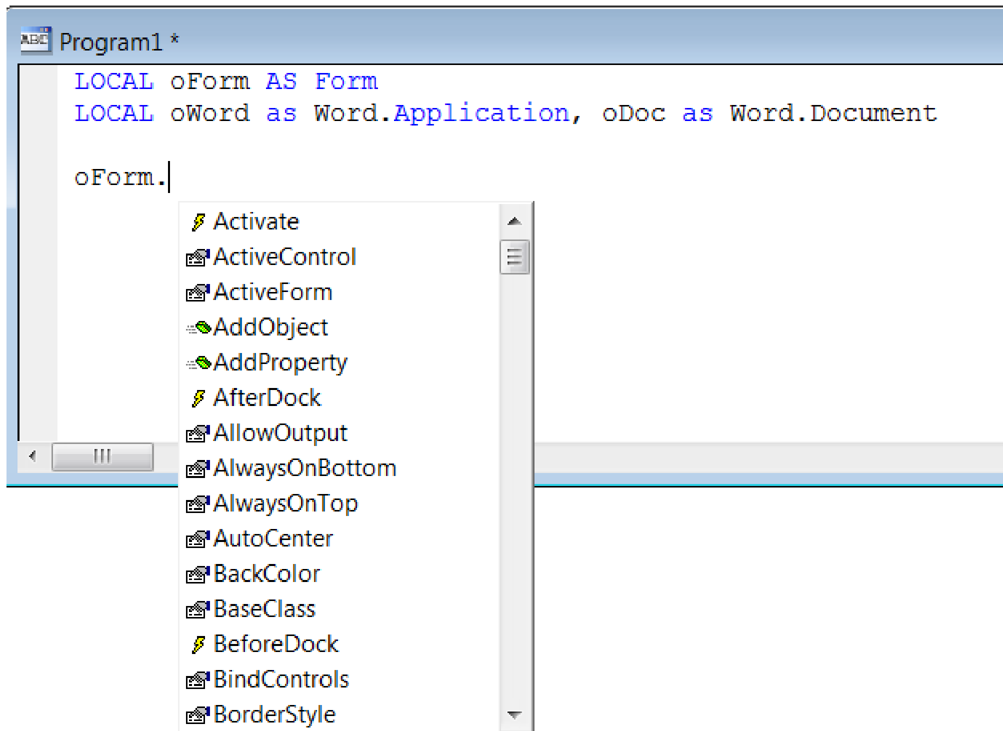

Figure 23. If you declare a variable as an object type that VFP knows about, you get IntelliSense when you type the variable name followed by a period.

You're not limited to built-in classes. As **Figure 23** suggests, you can use this technique for Automation objects. In addition, you can declare a variable AS some custom type; if VFP can find the class definition, you get the PEM list. If necessary, you can help VFP find the class definition by adding the OF clause to the declaration, as in **Figure 24**.

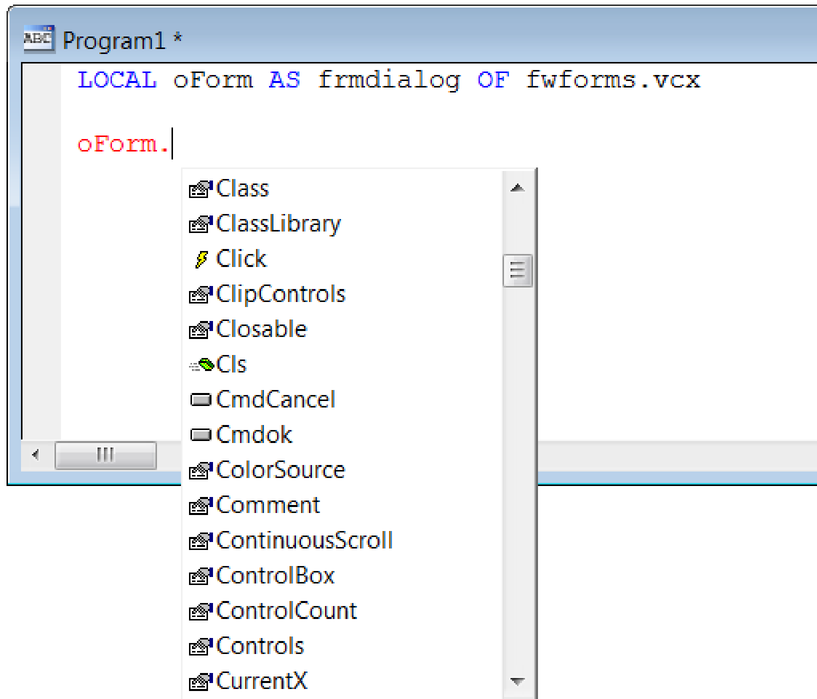

Figure 24. As long as VFP can actually find the relevant class library, this trick works for custom classes as well as native classes.

The same trick works to provide IntelliSense inside the WITH command. Use AS on the variable in the WITH command itself and when you type a period inside WITH, the list of PEMs appears, as in **Figure 25**.

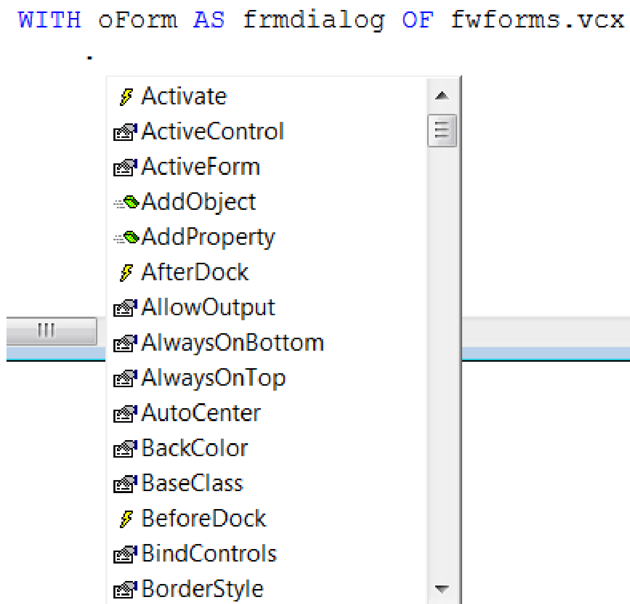

Figure 25. The AS keyword also lets you use IntelliSense inside a WITH clause.

Getting an object reference from a name

I often need to work with an object whose name is contained in a variable. For example, in one application, forms are populated on the fly, based on some meta-data. After adding a control, I need an object reference in order to set its properties.

The EVAL() function is perfect for this case. You can build a string that contains the exact path to the object you want and then call EVAL() to turn it into an object reference. For example, suppose cTextName contains the name you want to give a new textbox that you're adding to the form. **Listing 27** shows the code, assuming oForm holds a reference to the form you're working on. (A more complete example is included in the session materials as EvalForObjRef.PRG.)

Listing 27. You can build a reference to an object by sending its name to EVAL().

```
oForm.AddObject(m.cTextName, "textbox")
oTextBox = EVALUATE("oForm." + m.cTextName)
```

Odds and ends

There are a few bits of wisdom that don't fit into any of the categories above. So, they're grouped together here.

Never use TYPE XLS

The COPY TO command gives you a quick way to take data from a table or cursor and put it into another format. It offers many variations, and I could write an entire article about it.

But there's one issue I see over and over. One of the most common uses of COPY TO is creating Excel files. There are two variants that give you Excel output: TYPE XLS and TYPE XL5.

TYPE XLS creates an Excel 2.0 file, the type that was in use when COPY TO was added to FoxPro. Unless someone specifically asks for that format, don't use it. There was a significant change in date-handling in later versions of Excel. Use TYPE XL5 instead. While this is also an ancient format (the one before Excel 95), it's far more compatible with modern versions of Excel.

Be aware that even TYPE XL5 is limited to exporting 32,767 rows. Another option is to use TYPE CSV instead. CSV stands for comma-separated value, and Excel treats these as native files, as well. TYPE CSV doesn't have a limit on the number of rows you can export.

Use name expressions

There are quite a few places where VFP expects the name of something: a table, a field, a report, a file, etc. If you know the name of the thing, it's easy to just write the command: USE Customers.

But sometimes the name you want is in a variable or field. In the earliest days of FoxPro, the only way to handle that case was using the macro operator (&), like this: USE &cMyTable.

However, for a long time, there's been a much better alternative, called a *name expression*. What that means is surrounding the variable or field name with parentheses. VFP knows to evaluate what's in the parentheses and substitute that value. **Listing 28** shows an example; the user provides a file name to use with COPY TO.

Listing 28. Use a name expression rather than a macro anywhere VFP expects a name.

```
cXLFile = PUTFILE("File Name:", "Q3Sales", "XLS")
IF NOT EMPTY(m.cXLFile)
   COPY TO (m.cXLFile) TYPE XL5
ENDIF
```

You can use a name expression anywhere VFP expects a name and only a name. So, for example, in a SQL SELECT command, you can use a name expression for a table in the FROM clause and for the name of the cursor into which to put the results, but not where there might be multiple names, such as the field list or ORDER BY clause.

Name expressions are faster than macros, though on modern computers, the difference isn't generally enough to be visible. More importantly, name expressions work even when the name includes spaces. In Listing 28, a space in the file path works; if a macro were used, the code would fail in that case.

Check for arrays with TYPE()

The TYPE() function has been around for a long time, letting us check the type of fields, variables and other expressions. However, until VFP 9, TYPE() couldn't tell you whether a particular variable is an array. In VFP 9, a new parameter lets you ask that question. **Listing 29** demonstrates; note that you still must remember quotes around the name of the variable.

Listing 29. In VFP 9, TYPE() can tell you whether it's an array.

```
LOCAL nVal, cVal, aList[10,3]

nVal = 17
cVal = 'Tamar'

? TYPE('nVal') && N
? TYPE('cVal') && C
? TYPE('aList') && L because first element is logical

* Now, check for arrays
? TYPE('nVal', 1) && U
? TYPE('cVal', 1) && U
? TYPE('aList', 1) && A
```

Quick substitution for empty values

I used to write a lot of code that used IIF() or an IF statement to check whether a variable or field is empty and if so, substitute a default value for it, something like IIF(EMPTY(m.cVar), 'Default',m.cVar). Since VFP 8, I can shorten such code to EVL(m.cVar, 'Default').

The EVL() function works for empty values the way NVL() works for null values. In each case, the first parameter is evaluated. If it's not empty (for EVL()) or null (for NVL()), that value is returned. If it is empty or null, respectively, the second parameter is returned.

These functions work throughout VFP and I use them in many places, but they're particularly useful in parameter checking and in the field list of SQL SELECT, where they make code shorter and more readable. I have a lot of code that looks something like **Listing 30**, which comes from the Object Inspector.

Listing 30. This line from the Object Inspector's Init method shows how EVL() can shorten parameter checking.

```
This.cRootName = Evl(m.cRootName, This.cRootName)
```

NVL() has been around since VFP 6 and makes it easy to ensure that nulls don't propagate. Most often, I use NVL() to convert a null to an empty value for reporting.

Random string traps

VFP has two functions that generate random strings that you can use for various purposes, but each carries some risks. SYS(3) returns a random string of digits, while SYS(2015) returns a 10-chararacter string , always starting with an underscore ("_").

The problem with SYS(3) is that it repeats because the value returned is based on the system clock. If you call it several times in succession, you'll get the same value more than once. I used the code in **Listing 31** to test; it's included in the materials for this session as GeneratingRandomStrings.PRG. With 100,000 calls, I got 1800-2000 unique values from SYS(3). (Each time I ran the code, the result was slightly different.) SYS(2015), however, always generated 100,000 different strings.

Listing 31. SYS(3) doesn't return unique values, while SYS(2015) does. This code demonstrates.

```
CREATE CURSOR RandStrings (sys3 C(8), sys2015 C(10))

FOR nLoop = 1 TO 100000
   INSERT INTO RandStrings VALUES (SYS(3), SYS(2015))
ENDFOR

SELECT COUNT(distinct sys3) ;
   FROM randstrings ;
   INTO CURSOR csrSys3Count

SELECT COUNT(distinct sys2015) ;
   FROM randstrings ;
```

```
   INTO CURSOR csrSys2015Count
```

SYS(3) was never guaranteed to be unique, but the problem has gotten worse as computers have gotten faster. When I tested in the mid-90's with a Pentium/90, I got 134 different values with 1,000 calls. About 10 years later, I got between 1,100 and 1,200 unique values with 10,000 calls. Today, as noted, I'm getting only 1,800-2,000 unique values in 100,000 calls.

SYS(2015), on the other hand, is guaranteed to return a unique value on the given machine. The problem with SYS(2015) is much more subtle and less likely to bite you. The function was added to produce unique names for procedures created by GenScrn.PRG and GenMenu.PRG, the code generators included with FoxPro 2.0 that converted screen designs and menu designs, respectively, into PRGs.

However, some people started using it to generate random filenames back in the days when the filestem was limited to 8 characters. Since the function returns 10 characters, and the variation from one call to the next happens in the right-hand characters, a lot of people wrote code using SUBSTR(SYS(2015),3) for filenames.

That worked for a long time, but during July, 2003, SYS(2015) started returning values with a digit in the third position rather than a letter, and code started breaking.

Since filenames are no longer restricted to 8.3 format, the solution is to use the whole return value, or put a fixed string in front of some subset taken from the right.

One further note. I've seen people use these functions (especially SYS(3)) to generate unique aliases for cursors. In most situations, that's unnecessary. Cursors only need to be unique within the specified data session of the specified instance of the application. In almost every case, using a fixed name will do.

For those cases where you really do need to generate aliases, it's wise to ensure uniqueness by looping until you get a unique name from SYS(3), as in **Listing 32**.

Listing 32. If you need to use SYS(3) for multiple unique values, loop until you get them.

```
LOCAL cAlias
cAlias = "csr" + SYS(3)

DO WHILE USED(m.cAlias)
   cAlias = "csr" + SYS(3)
ENDDO
```

Final thoughts

The tips, tricks and traps in this article are just a sample of the thousands of ways you can work more efficiently or write better code with VFP. As noted earlier, I've focused only on what comes "in the box." The surest way to make yourself more efficient is to visit VFPX

and explore its many projects. If you haven't already installed Thor and GoFish, you're missing a lot of chances to save time while you work.

Copyright 2022, Tamar E. Granor