# New OOP and Language Features in Visual FoxPro 9.0

Doug Hennig

**VFP 9 has an incredible list of new features: dockable forms, control anchoring, polygon shapes, word wrapping for checkboxes and option buttons, IntelliSense support within WITH/ENDWITH and within memo fields, removal of limits in such places as the number of elements in an array or size of PRG files, new data types ... the list seems endless! This session will focus on new and improved OOP and language features so you can quickly get up to speed on the newest version of VFP.**

VFP 8 was touted as the biggest upgrade to VFP since its release. However, in many ways, I think VFP 9 is an even bigger upgrade. The reporting engine alone underwent a major uplift: new dialogs, multiple details bands, GDI+ rendering, new types of output, and open architecture are just some of the improvements in VFP 9. There are few places in VFP that weren't touched in this new version.

Because VFP is a programming language in addition to a database engine, this document concentrates on the improvements in the language and object features in VFP 9. However, no one white paper could cover all of the enhancements in this area. I won't discuss the following topics because they're covered by other white papers:

- SQL enhancements: see Tamar Granor's "Using SQL in Visual FoxPro 9.0."
- Data enhancements: David Anderson has two white papers on this topic.
- Reporting system enhancements: there are a variety of source for this, including documents by Cathy Pountney.
- Improvements to IntelliSense: Toni Feltman covers this and the new auto-completion feature in her "Using IntelliSense and AutoComplete in VFP Applications" white paper.

## New and enhanced language features

Every version of VFP adds new commands and functions and enhances the existing ones. VFP 9 is no exception.

### Fewer limits

One of the design goals of VFP 9 was to eliminate or at least raise the limits of earlier versions. For example, arrays can now have more than 64K elements. Given VFP's powerful cursor engine, why would you ever need an array of more than 65,535 elements? There may be several valid reasons, but VFP guru Rick Schummer told me about one he ran into: ADIR(). He was asked by a client to process a directory of images in some way. It turns out the directory had more than 14,000 images in it (why you'd do that is a valid question, but it wasn't Rick's choice). ADIR() creates an array with five columns, and 14,000 x 5 is more than 65,535, so his program bombed on the ADIR() statement. Fortunately, VFP 9 was in beta by this point, so he ran the program in that version without problems. Of course, you are limited by the amount of memory available, but that limit is extremely high these days.

Program and procedure size limits have been removed. Previously, the compiled size of a program or procedure was limited to 64K. Again, in a powerful OOP system such as VFP, you have to wonder why someone would design a system that pushed these limits, but one common place I'm aware of is GenDBC. This utility, located in the Tools\GenDBC subdirectory of the VFP home directory, creates a PRG that regenerates a database and all of its contents. A very large database could result in a PRG that compiles to more than 64K in size, so this is a legitimate issue for some people.

VFP 9 also lifts the former limit of 128 levels on the call stack. The new STACKSIZE setting in CONFIG.FPW allows you to adjust the nesting level from 32 to 64K deep. Once again, other than highly recursive code, I can't see why this was an issue for VFP developers, but perhaps it was an easy thing to do so the VFP team just went ahead with it.

### Improved string handling

VFP has always had fast yet powerful string handling functions. VFP 9 improves some of these functions even further.

You can now specify what characters the TRIM functions—TRIM(), RTRIM(), LTRIM(), and ALLTRIM()—remove from a string. These functions accept new parameters: a numeric value indicating case-sensitivity (0 or omitted for case-sensitive, 1 for case-insensitive) and one or more characters to remove. For example, in the past, we used to write code like the following to strip all white space (spaces, tabs, carriage returns, and line feeds) from the start and end of a string while preserving any occurrences in the middle:

```
do while inlist(left(lcString, 1), chr(13), chr(10), chr(9), ' ')
  lcString = substr(lcString, 2)
enddo
do while inlist(right(lcString, 1), chr(13), chr(10), chr(9), ' ')
  lcString = left(lcString, len(lcString) - 1)
enddo
```

In VFP 9, this simply becomes:

```
lcString = alltrim(lcString, 0, chr(13), chr(10), chr(9), ' ')
```

There are a couple of changes in ALINES(). First, the third parameter, which used to indicate whether the lines should be trimmed (.T.) or not (.F.), can now accept a numeric value. See **Table 1** for a list of the values, which are additive. The second change is that if ALINES() is used on a Varbinary or Blob data, the resulting array contains Varbinary elements.

*Table 1. The values for the third parameter to ALINES().*

| Value | Description |
|---|---|
| 1 | For character values, remove leading and trailing spaces from lines. For Varbinary and Blob values, remove trailing zeroes. This is the same as passing .T. (which you can still do for backward compatibility) in earlier versions. |
| 2 | Include the last element in the array even if it's empty. |
| 4 | Omit empty elements in the array. |
| 8 | Use case-insensitive parsing. |
| 16 | Include the parsing characters in the array. |

Here's an example that shows the differences between using 2 and 4 for the third parameter. There are four lines in the array when no parameter is specified, five when 2 is used, and three when 4 is used.

```
text to lcText noshow
Hi
DevCon

attendees

endtext
messagebox(transform(alines(laLines, lcText)) + ' lines with no parameter' + chr(13) + ;
  transform(alines(laLines, lcText, 2)) + ' lines with "include last line" (2)' + chr(13) + ;
  transform(alines(laLines, lcText, 4)) + ' lines with "no empty elements" (4)' + chr(13))
```

In the next example, the elements only have the closing brackets when 16 is specified:

```
lcText = '<html><body>This is some text</body></html>'
lcMessage = ''
for lnI = 1 to alines(laLines, lcText, '>')
  lcMessage = lcMessage + laLines[lnI] + chr(13)
next lnI
messagebox('Without "include parsing characters" (16):' + chr(13) + chr(13) + lcMessage)
lcMessage = ''
for lnI = 1 to alines(laLines, lcText, 16, '>')
  lcMessage = lcMessage + laLines[lnI] + chr(13)
next lnI
messagebox('With "include parsing characters" (16):' + chr(13) + chr(13) + lcMessage)
```

The TEXT command has a new clause, FLAGS, that customizes the behavior of the command. **Table 2** shows the values to specify in this clause; these values are additive.

*Table 2. The values for the FLAGS clause of the TEXT command.*

| Value | Description |
|---|---|
| 1 | Suppress output to the file specified in _TEXT. |
| 2 | Start the output with a carriage return and line feed. |

The following code shows the effect of these values. The first message shows “This is test 1This is test 2” on one line and “This is test 3” on another line (it was preceded by a carriage return and line feed because 2 was included in the FLAGS value) and the second message shows that “This is test 3” was not sent to the output file because 1 was included in the value.

```
_text = fcreate('somefile.txt')
text to lcText noshow
This is test 1
endtext

text to lcText additive noshow
This is test 2
endtext

text to lcText additive noshow flags 3
This is test 3
endtext

fclose(_text)
messagebox('The contents of the string are:' + chr(13) + chr(13) + lcText)
messagebox('The contents of the file are:' + chr(13) + chr(13) + filetostr('somefile.txt'))
```

VFP 9 supports a new value in the last parameter (the flags parameter) for STREXTRACT(): add 4 to include the delimiters in the return value. This is particularly handy when parsing HTML or XML, since previously you had to manually add the delimiters back to the expression. In the following example, the return value of the first STREXTRACT() doesn’t include “<img ” or “>” but the second one does.

```
lcText = getfile('HTM,HTML')
messagebox('Without "include delimiters" (4) flag:' + chr(13) + chr(13) + ;
  strextract(lcText, '<img ', '>', 1) + chr(13) + chr(13) + ;
  'With "include delimiters" (4) flag:' + chr(13) + chr(13) + ;
  strextract(lcText, '<img ', '>', 1, 4))
```

## Object-related functions

Developers who write design-time tools will like this new feature: passing 0 for the third parameter to NEWOBJECT() causes the specified object to be created without firing any initialization code (that is, code in events such as Init, Load, Activate, and BeforeOpenTables). When the object is destroyed, no destructor code (such as in Destroy or Unload) is fired. This is typically needed when you want to examine the structure of a class using AMEMBERS() (which means you have to instantiate it) but don’t want its normal behavior to occur. For example, suppose the following class exists in TestClass.PRG:

```
define class Noisy as Custom
  function Init
    messagebox('In init')
  endfunc
  function Destroy
    messagebox('In destroy')
  endfunc
enddefine
```

The following code won’t display the message boxes from the Noisy class, only the one in this code:

```
loObject = newobject('Noisy', 'testclass.prg', 0)
messagebox('Noisy has ' + transform(amembers(laMembers, loObject, 1)) + ' members')
release loObject
```

AGETCLASS() and PEMSTATUS(, 0) (which tells you if the specified property was changed) now operate in the VFP run-time. I haven't found a need to use AGETCLASS() at run-time, but I'm sure some tool developers have. One other PEMSTATUS() change: PEMSTATUS(, 5) now returns .F. for hidden native properties. Visually subclassable objects, such as DataEnvironment, Relation, and CursorAdapter, inherit from an internal RECT class, so they have properties defined in that class, such as Visible, Top, Left, and so forth, that aren't applicable. VFP hides these properties in the Properties window and IntelliSense, and AMEMBERS() doesn't see them, but in earlier versions of VFP, PEMSTATUS(, 5) returned .T. The VFP 9 version now returns .F.

Microsoft added the Collection base class in VFP 8, and it's been a wonderful addition except for one peculiarity: if you add objects to a collection, the FOR EACH command returns COM objects rather than VFP objects. This causes problems with functions such as AMEMBERS() and COMPOBJ(), as the following code illustrates:

```
loCollection = createobject('Collection')
loCollection.Add(createobject('Custom'))
for each loObject in loCollection
  messagebox('The object has '+ transform(amembers(laMembers, loObject, 1)) + ;
    ' members as a COM object')
next loObject
for each loObject in loCollection foxobject
  messagebox('The object has '+ transform(amembers(laMembers, loObject, 1)) + ;
    ' members as a VFP object')
next loObject
```

The first message shows zero members for the object. That's obviously not right, but has to do with the fact that loObject isn't a VFP Custom object but has been recast as a COM object. The second message displays the correct value; because of the new FOXOBJECT keyword in the FOR EACH command, loObject is a real VFP object.

CLEAR CLASSLIB has been enhanced in VFP 9 to implicitly do a CLEAR CLASS for each class within the specified class library.

## Windows message events

Windows communicates events to applications by passing them messages. Although VFP exposes some of these messages through events in VFP objects, such as MouseDown and Click, many messages are not available to VFP developers. One common request for this is the ability to detect an application switch. For example, I created an application that hooks into GoldMine, a popular contact management system, displaying additional information about the current contact. If the user switches to GoldMine, moves to a different contact, and then switches back to my application, it would be nice to refresh the display so it shows information about the new contact. Unfortunately, there was no way to do this in earlier versions of VFP; I had to rely on a timer that constantly checked which contact was currently displayed in GoldMine.

VFP 9 extends the BINDEVENT() function added in VFP 8 to support Windows messages. The syntax for this use is:

```
bindevent(hWnd, nMessage, oEventHandler, cDelegate)
```

where hWnd is the Windows handle for the window that receives events, nMessage is the Windows message number, and oEventHandler and cDelegate are the object and method that's fired when the message is received by the window. Unlike with VFP events, only one handler can bind to a particular hWnd and nMessage combination. Specifying a second event handler object or delegate method causes the first binding to be replaced with the second. VFP doesn't check for valid hWnd or nMessage values; if either is invalid, nothing will happen because the specified message can't be received by the specified window.

For hWnd, you can specify _Screen.hWnd or _VFP.hWnd to trap messages sent to the application or a form's hWnd for those messages sent to the form. VFP controls don't have a Windows handle, but ActiveX controls do, so you also bind to them.

There are literally thousands of Windows messages; search http://msdn.microsoft.com for a reference to the available messages. Example of such messages are: WM_POWERBROADCAST (0x0218), which is sent when a power event occurs such as low battery or switching to standby mode; WM_THEMECHANGED (0x031A), which indicates the Windows XP theme has changed; and WM_ACTIVATE (0x0006), which is raised when switching to or from an application.

The event handler method must accept four parameters: hWnd, the handle for the window that received the message, nMessage, the Windows message number, and two Integer parameters, the contents of which vary depending on the Windows message. The method must return an Integer, which contains a result value. One of the return values is BROADCAST_QUERY_DENY (0x424D5144, which represents the string “BMQD”), which prevents the event from occurring.

If you want the message to be processed in the normal manner, you’ll have to call the VFP Windows message handler in the event handler method; this is sort of like using DODEFAULT() in VFP method code. Here’s an example of an event handler that does this:

```
lparameters hWnd, ;
  Msg, ;
  wParam, ;
  lParam
local lnOldProc, ;
  lnResult
#define GWL_WNDPROC -4
declare integer GetWindowLong in Win32API ;
  integer hWnd, integer nIndex
declare integer CallWindowProc in Win32API ;
  integer lpPrevWndFunc, integer hWnd, integer Msg, integer wParam, ;
  integer lParam
lnOldProc = GetWindowLong(_screen.hWnd, GWL_WNDPROC)
lnResult  = CallWindowProc(lnOldProc, hWnd, Msg, wParam, lParam)
return lnResult
```

To determine which messages are bound, use AEVENTS(ArrayName, 1). It fills the specified array with one row per binding and four columns, containing the values of the parameters passed to BINDEVENT().

You can unbind events using UNBINDEVENT(hWnd [, nMessage ]). Omitting the second parameter unbinds all messages for the specified window. Passing only 0 unbinds all messages for all windows. Events are also automatically unbound when the event handling object is destroyed.

The VFP team added three SYS() functions related to Windows events in VFP 9. SYS(2325, wHandle) returns the wHandle for the client window whose wHandle is passed as a parameter (wHandle is an internal VFP wrapper for hWnd). SYS(2326, nWnd) returns the wHandle for the window specified with hWnd. SYS(2327, wHandle) returns the hWnd for the window specified with wHandle. The documentation for these functions indicates that they’re for BINDEVENT() scenarios using the VFP API Library Construction Kit. However, they can also be used to get the hWnd for the client window of a VFP IDE window. (A client window is a window inside a window; for example, _Screen is a client window of _VFP.) The FindIDEClientWindow method of the IDEWindowsEvents class in TestWinEventsForIDE.PRG shows an example of the use of these functions.

TestWinEventsForIDE.PRG demonstrates event binding to VFP IDE windows. Set lcCaption to the caption of the IDE window you want to bind events to, run the program, and activate and deactivate the window, move it, resize it, and so forth. You should see Windows events echoed to the screen. When you’re finished, type RESUME and press Enter in the Command window to clean up. To test this with the client window of an IDE window, uncomment the indicated code. You can, of course, try binding to other events as well by adding BINDEVENT() statements to this code; use the constants in WinEvents.H for the values for the desired events.

```
#include WinEvents.H

lcCaption      = 'Command'
loEventHandler = createobject('IDEWindowsEvents')
lnhWnd         = loEventHandler.FindIDEWindow(lcCaption)
* Uncomment this code to receive events for the window's client window instead
*lnhWnd         = loEventHandler.FindIDEClientWindow(lcCaption)
if lnhWnd > 0
  bindevent(lnhWnd, WM_SETFOCUS,      loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_KILLFOCUS,     loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_MOVE,          loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_SIZE,          loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_MOUSEACTIVATE, loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_KEYDOWN,       loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_KEYUP,         loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_CHAR,          loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_DEADCHAR,      loEventHandler, 'EventHandler')
  bindevent(lnhWnd, WM_KEYLAST,       loEventHandler, 'EventHandler')
  clear
  suspend
  clear
else
  messagebox('The ' + lcCaption + ' window was not found.')
endif lnhWnd > 0

define class IDEWindowsEvents as Custom
  cCaption = ''
  nOldProc = 0

  function Init
    declare integer GetWindowLong in Win32API ;
      integer hWnd, integer nIndex
    declare integer CallWindowProc in Win32API ;
      integer lpPrevWndFunc, integer hWnd, integer Msg, integer wParam, ;
      integer lParam
    declare integer FindWindowEx in Win32API;
      integer, integer, string, string
    declare integer GetWindowText in Win32API ;
      integer, string @, integer
    This.nOldProc = GetWindowLong(_screen.hWnd, GWL_WNDPROC)
  endfunc

  function FindIDEWindow(tcCaption)
    local lnhWnd, ;
      lnhChild, ;
      lcCaption
    This.cCaption = tcCaption
    lnhWnd        = _screen.hWnd
    lnhChild      = 0
    do while .T.
      lnhChild = FindWindowEx(lnhWnd, lnhChild, 0, 0)
      if lnhChild = 0
        exit
      endif lnhChild = 0
      lcCaption = space(80)
      GetWindowText(lnhChild, @lcCaption, len(lcCaption))
      lcCaption = upper(left(lcCaption, at(chr(0), lcCaption) - 1))
      if lcCaption = upper(tcCaption)
        exit
      endif lcCaption = upper(tcCaption)
    enddo while .T.
    return lnhChild
  endfunc

  function FindIDEClientWindow(tcCaption)
    local lnhWnd, ;
      lnwHandle, ;
      lnwChild
    lnhWnd = This.FindIDEWindow(tcCaption)
    if lnhWnd > 0
      lnwHandle = sys(2326, lnhWnd)
      lnwChild  = sys(2325, lnwHandle)
      lnhWnd    = sys(2327, lnwChild)
    endif lnhWnd > 0
    return lnhWnd
  endfunc

  function EventHandler(hWnd, Msg, wParam, lParam)
    ? 'The ' + This.cCaption + ' window received event #' + transform(Msg)
    return CallWindowProc(This.nOldProc, hWnd, Msg, wParam, lParam)
  endfunc
enddefine
```

WindowsMessagesDemo.SCX is another example. It shows hooking into activate and deactivate events as well as certain Window shell events, such as inserting or removing a CD or USB drive. The latter shows an interesting use of Windows events: the code registers _VFP to receive a subset of Windows shell events as a custom Windows event. The following code, taken from the Init method of this form, handles the necessary setup. Items in upper-case are constants defined in either WinEvents.H or ShellFileEvents.H. The call to SHChangeNotifyRegister tells Windows to register _VFP to receive disk events, media insertion and removal events, and drive addition and removal events using a custom message. This code then binds device change events and the custom message we just defined to the HandleEvents method of the form.

```
declare integer SHChangeNotifyRegister in shell32 ;
  integer hWnd, integer fSources, integer fEvents, integer wMsg, ;
  integer cEntries, string @SEntry

* Register us to receive certain shell events as a custom Windows event.

lcSEntry = replicate(chr(0), 8)
This.nShNotify = SHChangeNotifyRegister(_vfp.hWnd, SHCNE_DISKEVENTS, ;
  SHCNE_MEDIAINSERTED + SHCNE_MEDIAREMOVED + SHCNE_DRIVEADD + ;
  SHCNE_DRIVEREMOVED, WM_USER_SHNOTIFY, 1, @lcSEntry)

* Bind to the Windows events we're interested in.

bindevent(_vfp.hWnd, WM_DEVICECHANGE,  This, 'HandleEvents')
bindevent(_vfp.hWnd, WM_USER_SHNOTIFY, This, 'HandleEvents')
```

Support for Windows event binding is an incredible addition to VFP; it allows us to hook into just about anything that goes on in VFP. I expect we'll see lots of cool uses of this as the VFP community starts to learn what its capabilities are.

## Internationalization issues

FoxPro has had strong support for internationalization issues since the 2.x days added support for code pages and collate sequences. VFP 9 adds additional support.

The fourth parameter for GETFONT() indicates what language script to display by default. In VFP 8, 0 means you want the Script combo box in the Font dialog disabled, while 1 means to select Western by default. The problem is that unless you know the default language script setting for the user, what value should you pass for this parameter? In VFP 9, 0 now means Western and 1 means the user default script.

STRCONV() can now accept a locale ID, code page, or FontCharSet value for the third parameter (previously, only a locale ID was supported). A new fourth parameter indicates which type is used: 0 or omitted means locale ID, 1 means code page, and 2 means FontCharSet. Unfortunately, I don't have much knowledge about internationalization issues, so I can't provide a meaningful example.

DEFINE POPUP has new RTLJUSTIFY and LTRJUSTIFY clauses that tell VFP to justify the text in a right-to-left or left-to-right manner. The clause is ignored unless Windows is configured to a Middle-Eastern locale. You can also set the justification for ToolTips using the new SYS(3009) function. SYS(3009, 1) specifies right-to-left and SYS(3009) or SYS(3009, 0) specifies left-to-right. This function returns the current value as a character.

## Other enhanced commands and functions

SET PATH has a new ADDITIVE clause that allows you to add to the existing VFP path. Formerly, you had to do something like:

```
lcCurrPath = set('PATH')
set path to &lcCurrPath, C:\MyNewDirectory
```

Now it's as simple as:

```
set path to 'C:\MyNewDirectory' additive
```

Note that quotes are required around the path even if there aren't any spaces in it when you use the ADDITIVE clause; if you omit them, the path becomes "C:\MyNewDirectory ADDITIVE." The size of the path has also been increased from 1024 to 4095 characters.

The TYPE() function has a new parameter: specify 1 to determine if the variable is an array or collection; TYPE() returns “A” if it’s an array, “C” if it’s a collection, or “U” if neither. For example:

```
X = 1
dimension Y[1]
? type('X', 1)           && displays "U"
? type('Y')              && displays "L", the data type for Y[1]
? type('Y', 1)           && displays "A"
loObject = createobject('Collection')
? type('loObject')       && displays "O"
? type('loObject', 1)    && displays "C"
```

You can now specify what value to return if the user clicks on the Cancel button or presses Esc in the INPUTBOX() function by specifying a value for the new sixth parameter. In previous versions of VFP, this function returns a blank string. Here’s an example that demonstrates this; the message displays “Cancel value” when you click the Cancel button or press Esc.

```
messagebox(inputbox('Click on cancel', 'Test inputbox()', 'Default value', 0, ;
  'Timeout value', 'Cancel value'))
```

Three SYS() functions have been enhanced:

- SYS(1104), which purges cached memory, now supports an optional workarea or alias parameter. Specifying this tells VFP to purge the cache for the specified table or cursor.
- SYS(3056) can now write the settings of the Tools | Options dialog to the Registry if you add 2 to the optional parameter. This does the same thing as clicking on the Set As Default button in that dialog.
- VFP 8 added the ability to specify an external configuration file by adding ALLOWEXTERNAL = ON to the configuration file built into an EXE. SYS(2019) returns the name of the external configuration file if you specify 2 as the optional parameter. Specify 1 or omit the parameter to return the name of the internal configuration file.

The FFLUSH() function, used to write out changes to a file opened with low-level file functions to disk, has a new second parameter. Pass .T. to force VFP to immediately write the file to disk. If you omit this parameter or pass .F., VFP will write the file when it gets a chance to.

Remember the SET DOHISTORY command? Okay, I don’t either <g>. It’s a function that outputs lines of code as they execute. In VFP 9, the output now goes to the Debug Output window, if it’s open, rather than to the current output device (such as _Screen or a printer if SET PRINT ON is used).

You can now pass 0 to MROW() and MCOL() to return the position of the mouse pointer based on the active form. This prevents a problem using these functions when the AllowOutput property of the form is .F.: they normally reference the form whose name is returned by WOUTPUT(), but WOUTPUT() doesn’t return the name of the form when AllowOutput is .F.

## New commands and functions

ICASE() is one of my favorite new functions. It’s similar to IIF(), except it acts like a DO CASE structure rather than an IF structure. This function saves having to use nested IIF() calls when you need an expression that selects between more than two values. Field expressions in the Report Designer are a common use for this. For example, rather than:

```
iif(SHIPVIA = 1, 'Fedex', iif(SHIPVIA = 2, 'UPS', iif(SHIPVIA = 3, 'DHL', 'Mail')))
```

you can use:

```
icase(SHIPVIA = 1, 'Fedex', SHIPVIA = 2, 'UPS', SHIPVIA = 3, 'DHL', 'Mail')
```

Parameters go in pairs. The first of a pair is the expression to evaluate and the second is the value to return if the expression is .T. If the last parameter isn’t one of a pair (that is, there’s an odd number of parameters), it’s considered to be the “otherwise” value, such as “Mail” is in the preceding example.

CAST(), another of my favorite new functions, allows you to convert data from one type into another (within reason, of course; Date fields cannot be converted to Numeric and vice versa). The syntax is:

```
CAST( uExpression AS DataType [ ( nFieldWidth [, nPrecision ] ) ] [ NULL | NOT NULL ] )
```

uExpression is the expression to convert and DataType is the data type to convert it to. DataType can be specified as a letter (for example, "T" for DateTime") or a name (VFP 9 supports long names for data types, such as "Numeric"). As in the CREATE TABLE command, some data types require a field width and possibly precision. You can also indicate whether nulls values are allowed or not.

CAST() can be used anywhere you'd use an expression, even in SQL SELECT statements. Here's a simple example that converts a DateTime field in a table into a Date value in a cursor:

```
select ORDERID, cast(ORDERDATE as Date) as ORDERDATE from ORDERS
```

CAST() can be used to force a SQL SELECT statement to create a field of the correct data type and size from an expression. For example, this SQL SELECT statement:

```
select UNITPRICE * QUANTITY as TOTAL, ;
  iif(UNITPRICE * QUANTITY >= 1000, 50, 0) as SHIPPING ;
  from ORDERDETAILS
```

works properly if the first record has a total amount of $1,000 or more, but not if the amount is less. That's because VFP uses the first record to determine the data type and size of the column. If the amount is less than $1,000, the expression returns 0, so the SHIPPING column will be N(1, 0). As a result, it isn't big enough for 50, and displays asterisks instead. A trick that helps with this is to specify the smallest number using a placeholder with the proper number of digits, such as 00, but that may not work in every situation (such as when you call a user-defined function) and can't help if you want a different data type, such as Currency, than VFP thinks you need.

If you change the SQL SELECT statement to:

```
select UNITPRICE * QUANTITY as TOTAL, ;
  cast(iif(UNITPRICE * QUANTITY >= 1000, 50, 0) as Numeric(5, 2)) as SHIPPING ;
  from ORDERDETAILS
```

or even:

```
select UNITPRICE * QUANTITY as TOTAL, ;
  cast(iif(UNITPRICE * QUANTITY >= 1000, 50, 0) as Currency) as SHIPPING ;
  from ORDERDETAILS
```

the values in the first record are unimportant; SHIPPING will always be the desired data type and size.

What would a new version of VFP be without new SYS() functions? SYS(2910) allows you to determine or set the number of items that appear in IDE (Interactive Development Environment) list boxes, such as IntelliSense, and in the new AutoComplete feature for text boxes which I'll discuss later. This corresponds to the List display count setting in the View page of the Tools | Options dialog. The range is 5 to 200; the default is 15. SYS(2910) by itself returns the current value as a character string.

There are two new features that provide better support for Tablet PCs. The ISPEN() function returns .T. if the last mouse event was a pen tap. A logical place to use this is in the Click method of a control if you want to perform different behavior for mouse clicks and pen taps. A new property of _Screen, DisplayOrientation, indicates the display orientation. The values for this property are shown in **Table 3**. Interestingly, this property is read-write; changing its value causes the display orientation to change as if you changed it from the Control Panel. See Markus Egger's "Building Tablet PC Applications with VFP" white paper for more information on this topic.

*Table 3. The return values for _Screen.DisplayOrientation.*

| Value | Description |
|---:|---|
| 0 | Upright landscape |
| 1 | Upright portrait |
| 2 | Inverted landscape |

| | |
|---|---|
| 3 | Inverted portrait |

The new CLEAR ERROR command wipes out all VFP error information as if no error had occurred. AERROR() returns 0 and doesn't create the array if the one specified doesn't exist, ERROR() returns 0, and MESSAGE(), MESSAGE(1), and SYS(2018) return blank strings.

## New and enhanced OOP features

There are four big new object-related features in VFP 9 and lots of smaller, fine-tuning ones.

### Anchoring visual controls

Nothing makes an application look more unprofessional than resizable forms in which nothing moves or resizes. If you've written manual resizing code, you know what an ugly chore this is, not to mention a maintenance nightmare. Since VFP 6, the FFC (FoxPro Foundation Classes) subdirectory of the VFP home directory has included an _Resizable class to automate this chore. However, it wasn't perfect, and I've written a couple of articles in the past on better mechanisms for resizing.

Fortunately, VFP 9 makes all of that superfluous: all visual controls support anchoring, which determines how a control behaves when its container resizes. The reason this is called anchoring is because you can think of it as anchoring the edges or center of a control to one or more borders of its container. For example, if a control is anchored to the right border, it moves horizontally as the container is resized so it maintains a constant distance from that border. If a control is anchored to both the top and bottom borders, it doesn't move but instead resizes vertically as the container is resized, since both its top and bottom edges maintain a constant distance from their respective container borders.

Anchoring a control is easy: simply set the Anchor property to the desired value. Figuring out the desired value, on the other hand, can be a bit of a challenge. **Table 4** shows the values to use for a particular anchoring effect; these values are additive, so 15 (1 + 2 + 4 + 8) causes a control to resize vertically and horizontally.

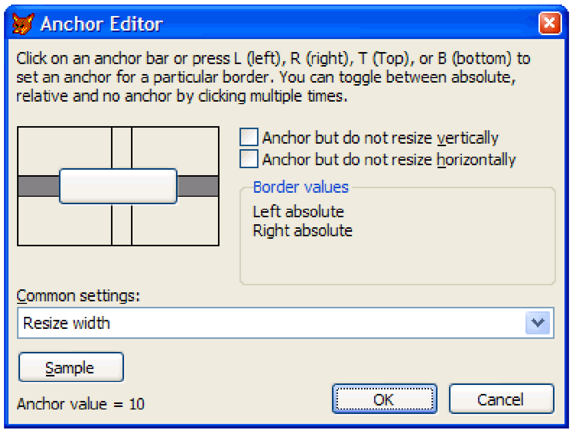
"Relative" anchoring is used when two or more controls resize in such as way that they'd conflict. For example, two side-by-side edit boxes with Anchor set to 10 (2 + 8, anchored to the left and right borders) will eventually overlap as the form is widened. Setting Anchor to 160 instead (32 + 128, relatively anchored to the left and right borders) causes these controls to resize by only half the amount of the form width increase (or a third if there are three controls, and so forth), so they won't overlap. Note that you can't specify both absolute and relative anchoring to the same border (such as 17, 1 + 16) or you'll get an error.

The last two values, 256 and 512, are typically used to keep a control, often a command button, centered on the form. These values conflict with other values. For example, you can't anchor to the right or left borders while at the same time anchoring the horizontal center; in other words, 2 and/or 8 conflict with 256.

*Table 4. The additive values for the Anchor property.*

| Value | Description |
|---|---|
| 0 | No anchoring (or anchored to the top and left borders if you'd rather think of it that way). |
| 1 | Anchored to the top border. |
| 2 | Anchored to the left border. |
| 4 | Anchored to the bottom border. |
| 8 | Anchored to the right border. |
| 16 | Relatively anchored to the top border. |
| 32 | Relatively anchored to the left border. |
| 64 | Relatively anchored to the bottom border. |
| 128 | Relatively anchored to the right border. |
| 256 | Anchors the horizontal center of the control to the left and right borders but doesn't resize. |
| 512 | Anchors the vertical center of the control to the top and bottom borders but doesn't resize. |

Setting anchor values manually would be extremely tedious, so VFP 9 comes with an Anchor Editor. This editor is registered as the property editor for the Anchor property, so simply select that property in the Properties window and click on the editor button (the one with the ellipsis in the upper area of the Properties window). **Figure 1** shows the Anchor Editor, although, since VFP 9 is still in beta at the time of writing, the user interface and/or behavior may change.


Anchor Editor

Click on an anchor bar or press L (left), R (right), T (Top), or B (bottom) to set an anchor for a particular border. You can toggle between absolute, relative and no anchor by clicking multiple times.

☐ Anchor but do not resize vertically
☐ Anchor but do not resize horizontally

Border values
Left absolute
Right absolute

Common settings: Resize width

Anchor value = 10

*Figure 1. The Anchor Editor makes it easy to set the Anchor property for all selected controls.*

## Dockable forms

Many of the VFP IDE windows are dockable: like toolbars, they can be docked at any edge of the screen and even be docked together. To make a window dockable, right-click in its title bar and turn on the Dockable setting. To dock a window, drag it to an edge of the screen until it docks. To dock it with another dockable window, drag it into the other window. To tab-dock it with another dockable window, drag its title bar onto the title bar of the other window.

Until VFP 9, VFP forms could not be dockable. That's changed with the addition of the Dockable property. The default value is 0-Not dockable. The other choice are 1 (supports docking and is dockable) and 2 (supports docking but is not currently dockable). With the latter two settings, a Dockable item appears in the form's title bar shortcut menu so a user can easily switch between these two settings. A dockable VFP form can be docked like any of the dockable IDE windows: to an edge of the form, to another dockable window or form, or tab-docked to another dockable window or form.

There are a few interesting things regarding dockable forms:

- The ScrollBars, ShowWindow, and Desktop properties must be set to 0-None, 0-In Screen, and .F., respectively, or Dockable can't be set to 1 or 2. This is true both at design time and run time.
- When Dockable is set to 1 or 2, the values of several other properties are automatically changed (at run time, not design time). For example, AlwaysOnBottom, AlwaysOnTop, MinButton, and MaxButton are set to .F., HalfHeightCaption is set to .T., WindowType is set to 0-Modeless (since a dockable window can't be modal), and Visible is set to .T.; see the VFP help topic for the Dockable property for a complete list of affected properties. You also aren't allowed to edit these properties at design time, even if their values aren't the ones used at run time.
- If you set Dockable to 1 or 2 in the Properties window, the form becomes visible before the Init method fires when the form is run. This may be a problem if, for example, you change something about the form dynamically based on parameters passed to Init. In that case, set Dockable in code when you're ready for the form to become visible rather than in the Properties window.
- When Dockable is set to 1 or 2, VFP disregards the settings of the MaxLeft, MaxTop, MaxWidth, MinHeight, and MinWidth properties and sets BorderStyle to 3-Sizable. This means the user can resize the form anyway they wish, such as making it ridiculously small.

Several properties, events, and methods formerly only used for toolbars now apply to forms as a result of this new capability: the Docked and DockPosition properties, the BeforeDock, AfterDock, and Undock events, and the Dock method. There's also a new method, GetDockState, which fills the specified array with information about the docking state of the form. Note that unlike functions that take arrays as parameters, such as the related ADOCKSTATE(), the array must already exists and you must pass its name as a string. **Table 5** shows the contents GetDockState puts into the array. This method currently has a number of bugs and odd issues that I hope will be resolved in the release version.

*Table 5. The contents of the array filled by GetDockState.*

| Column | Description |
|---|---|
| 1 | The caption of the form. The documentation says this is the name of the form, so I'm not sure if this is a product or documentation bug. I'm not sure why this is in the array, since if you can call the GetDockState, you must have a reference to the form and can easily access its Caption or Name property. |
| 2 | Dock state: 1 = docked, 2 = not docked. |
| 3 | Dock position: see the VFP help for DockPosition for a list of values. |
| 4 | The caption of the form this form is docked to, or empty if it isn't docked to another form. As with column 1, the documentation states this should be the name of the form. If the form is docked to one of the borders of the screen, this contains _Screen.Caption. |
| 5 | A reference to this form. As with column 1, I'm not sure why this is in the array, since you must have a reference to the form to call GetDockState. |
| 6 | A reference to the form this form is docked to, or an empty string if it isn't docked to another form (in my opinion, it should be .NULL. instead of a blank string). If the form is docked to one of the borders of the screen, this contains a reference to _Screen. |

## Auto-completion

Microsoft Internet Explorer has used auto-complete text boxes for quite a while. They show up in at least two places: the address bar and text boxes in HTML pages. For example, when you start typing the URL for a Web site, if you've visited any sites that start with the letters you're typing, a drop-down list of matching items appears, allowing you to quickly select one with the mouse or arrow keys. You can now add this feature to your VFP applications by setting the new AutoComplete property of a text box to the desired value (see **Table 6**).

*Table 6. The values for the AutoComplete property.*

| Value | Description |
|---|---|
| 0 | No auto-completion. |
| 1 | Supports auto-completion, alphabetical order |
| 2 | Supports auto-completion, most frequently used order |
| 3 | Supports auto-completion, most recently used order |
| 4 | Supports auto-completion, user-defined weighting order |

By default, entries the user makes into auto-completion text boxes go into a table called AutoComp.DBF in the VFP home directory. However, you can change the name and path of this table globally by setting _Screen.AutoCompTable or for a particular text box by setting its AutoCompTable property. VFP uses the text box name to distinguish the values entered in one text box from another, but you can set the AutoCompSource property to something more meaningful if you wish.

I'm not going to go any further on this topic because Toni Feltman's "Using IntelliSense and AutoComplete in VFP Applications" white paper discusses this in great detail.

## Creating complex shapes with PolyPoints

VFP 9 allows you to create complex shapes without using image files with the new PolyPoints property of Shape and Line controls. You can create polygon shapes, polygon lines, and Bezier curves with this feature.

PolyPoints contains the name of an array of polygon points. Typically this array will be two-dimensional since points are represented by X and Y values. The array must have a least two pairs of coordinates to draw a line, three pairs for a polygon line or shape, four pairs for a closed polygon shape, and 3n + 1 pairs for a Bezier curve, where n is the number of curves to draw. For a closed shape, the last pair of coordinates must match the first pair. To create a Bezier curve, set the LineSlant property to "s" (case is unimportant). Note that the coordinates are expressed as percentages of the object size rather than absolute values, so 50, 50 signifies a position at 50% of the height and 50% of the width.

The following code is taken from the Init method of PolyPointsDemo.SCX. It fills three array properties with the coordinates for three Line or Shape objects and then sets the PolyPoints property of these objects to the names of the arrays. **Figure 2** shows the resulting form. Note that PolyPoints doesn't affect the shape of the objects at design-time, only at run-time, so you won't see these shapes in the Form Designer. Also, the Click method of the triangle (which is actually a Line) increases the object's Height and Width by 25%. When you click on it, notice that the triangle gets larger without adjusting the array because the coordinates are expressed as percentages of the object's size.

```
with This
  dimension .poly1[6, 2]    && star
  .poly1[1, 1] = 50
  .poly1[1, 2] = 0
  .poly1[2, 1] = 100
  .poly1[2, 2] = 100
  .poly1[3, 1] = 0
  .poly1[3, 2] = 50
  .poly1[4, 1] = 100
  .poly1[4, 2] = 50
  .poly1[5, 1] = 0
  .poly1[5, 2] = 100
  .poly1[6, 1] = 50
  .poly1[6, 2] = 0
  .sfshape1.PolyPoints = 'Thisform.poly1'
  dimension .Poly2[4, 2]    && triangle
  .Poly2[1, 1] = 50
  .Poly2[1, 2] = 0
  .Poly2[2, 1] = 100
  .Poly2[2, 2] = 100
  .Poly2[3, 1] = 0
  .Poly2[3, 2] = 100
  .Poly2[4, 1] = 50
  .Poly2[4, 2] = 0
  .sfline2.PolyPoints = 'Thisform.poly2'
  dimension .Poly3[7, 2]    && Bezier curve
  .Poly3[1, 1] = 50
  .Poly3[1, 2] = 0
  .Poly3[2, 1] = 25
  .Poly3[2, 2] = 17
  .Poly3[3, 1] = 25
  .Poly3[3, 2] = 33
  .Poly3[4, 1] = 50
  .Poly3[4, 2] = 50
  .Poly3[5, 1] = 75
  .Poly3[5, 2] = 67
  .Poly3[6, 1] = 75
  .Poly3[6, 2] = 83
  .Poly3[7, 1] = 50
  .Poly3[7, 2] = 100
  .sfline1.PolyPoints = 'Thisform.poly3'
endwith
```

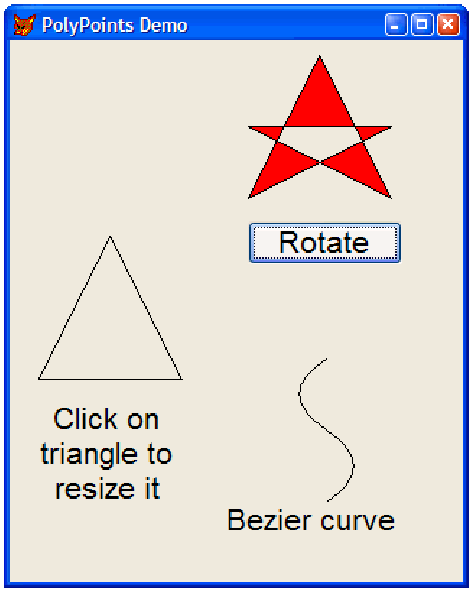


*Figure 2. The PolyPoints property of Line and Shape objects allow you to create complex shapes at run-time without using images.*

## Enhancements to existing properties and methods

VFP 9 extends the behavior of some existing properties.

- CheckBox and OptionButton controls now have a WordWrap property so you can make them narrower and word-wrap the text as necessary.
- The Alignment property for CheckBox and OptionButton supports more values (formerly, the choices were 0-Left and 1-Right), so you have more control over how they appear. The choices are 1, 2, and 3 for middle left, right, and center, 4, 5, and 6 for top left, right, and center, and 7, 8, and 9 for bottom left, right, and center.
- CommandButton now supports an Alignment property which affects both the alignment of the text and the alignment of the text and picture if a picture is used and PicturePosition is set to something other than the default of 13 (above caption, with caption centered along bottom).
- The PicturePosition property of CommandButton, OptionButton, and CheckBox supports a new value: 14 (no text). This is useful when you want a picture and no text, but you want a hotkey for the control, which means you have to use "\<" in the Caption property.
- Set the RowSourceType property of a ComboBox or ListBox to 10 to base the control's list of values on a collection. Set RowSource to the name of the collection object. If the collection consists of objects, add to RowSource a comma and the name of the property in the object to get the values from. For example, the following simple code, placed in the Init of a ListBox, creates a collection of objects with a property called CustomValue and uses that property as the source of values for the ListBox:

```
loCollection = createobject('Collection')
for lnI = asc('A') to asc('F')
  loObject = createobject('Empty')
  addproperty(loObject, 'CustomValue', chr(lnI))
  loCollection.Add(loObject)
next lnI
This.AddProperty('oCollection', loCollection)
This.RowSourceType = 10
This.RowSource = 'This.oCollection, CustomValue'
```

- The Style property of Labels supports a new value: 4, for themed background only. This is used when you want a themed label but want control over the foreground color; 3-Themed ignores the ForeColor property and uses the appropriate system color for the current theme.
- MousePointer and MouseIcon properties have been added to Column and Header objects, allowing you to control the appearance of the mouse pointer over these controls. This can be used, for example, as a visual indication that the user can click on a column to sort the grid on that column. Also, MousePointer has two new values: 16-Down Arrow and 17-Magnifying Glass.
- If the Style of a Separator in a Toolbar is set to 1, the Separator will now appear as a horizontal line when the Toolbar is oriented vertically. Previously, the Separator did not appear in that case. Also, Separators now have a Visible property; if it's set to .F., neither a space nor a line appear for the Separator.
- The InputMask property accepts two new characters: "U" permits only alphabetical characters and converts them to upper-case (similar to "!" but that allows non-alphabetical characters) and "W" permits only alphabetical characters and converts them to lower-case. A new "Z" character in the Format property specifies that a numeric value of zero or a blank date displays as blank except when the control has focus.
- Setting the AllowCellSelection property of a Grid to .F. now prevents a user from deleting a record by clicking on the delete mark.
- You can now prevent a ComboBox from being dropped down by putting NODEFAULT in the DropDown method.
- The Page Up, Page Down, Home, and End keys now fire the Click method of a ListBox.
- You still can't use SetFocus in Valid method of an object, but you can now return a reference to the object that should receive focus. For example, the following code in the Valid method of a text box allows you to set focus to different controls, depending on the value:

```
do case
  case This.Value = 1
    return Thisform.txtObject1
  case This.Value = 2
    return Thisform.txtObject2
  otherwise
    messagebox('Invalid value')
    return 0
endcase
```

## New properties

VFP 9 adds several new properties to objects.

- You can control the spacing between the picture and the text and between the picture and the edge of the control using the new PictureSpacing and PictureMargin properties of CommandButton, OptionButton, and CheckBox controls.
- You can rotate Label, Line, and Shape controls, even those that display complex shapes due to PolyPoints, by setting the new Rotation property to the desired value.
- Even if a ListBox isn't filled with items, its scrollbar appears unless you set the new AutoHideScrollBar property to 1 (scrollbar is only visible when listbox filled). You can also set it to 2 (scrollbar is never visible) if you wish.
- Image controls now have a PictureVal property. Unlike Picture, which contains the name of a file to use for the image, PictureVal contains the actual image itself. This combined with the new Blob data type is a General field killer. General fields suffer from lots of problems: hard to use, hard to update, subject to corruption, huge size, etc. Instead of storing images in a General field in a table, you can now either store the path to the images in a Character or Memo field or store the actual content of the images in a Blob field. You can then use code like this to display the image for the current record in the table:

```
This.imgPicture.PictureVal = filetostr(EMPLOYEES.PICTURE)  && if PICTURE contains filename
This.imgPicture.PictureVal = EMPLOYEES.PICTURE              && if PICTURE is Blob containing image
```

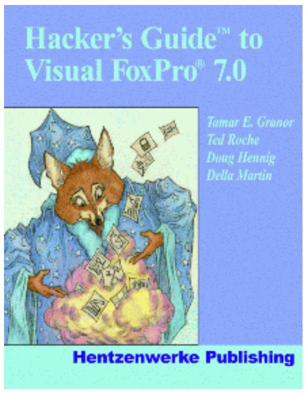
## Summary

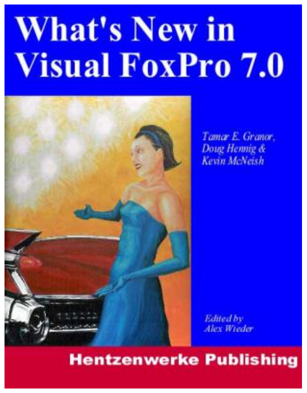
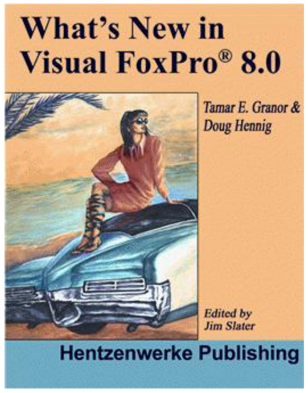
The language and object enhancements in VFP 9 make it easier than ever before to deliver the kinds of powerful applications our users expect. Some of them will make our code faster to write, faster to run, and more maintainable because we can write less code. Others allow us to do things we couldn't before, such as hooking into Windows events. Improvements in visual controls allow us to make more attractive and user-friendly applications. Overall, VFP 9 is a definite winner!

## Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), the award-winning Stonefield Query, and the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro. Doug is co-author of the "What's New in Visual FoxPro" series (the latest being "What's New in Visual FoxPro 9.0") and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). Doug writes the monthly "Reusable Tools" column in FoxTalk. He has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He has been a Microsoft Most Valuable Professional (MVP) since 1996.

*Doug Hennig*
*Stonefield Software Inc.*
*1112 Winnipeg Street, Suite 200*
*Regina, SK Canada S4R 1J6*
*Voice: 306-586-3341*
*Fax: 306-586-5080*
*Email: dhennig@stonefield.com*
*Web: www.stonefield.com*
*Web: www.stonefieldquery.com*




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Edited by Alex Wieder
Hentzenwerke Publishing

What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing