DEV180

Visual FoxPro Quirks

By Tamar E. Granor, Ph.D.
Editor, FoxPro Advisor

Visual FoxPro has many features that are confusing and could even be mistaken for bugs. This session looks at a bunch of those items, including the difference between the AddItem and AddListItem methods, why variables you create in the Command Window sometimes disappear, and much more. Find out why VFP behaves the way it does. In many cases, we'll look at the historical reasons why things behave as they do, as well.

Much of the material in those notes is excerpted from my books Hacker's Guide to Visual FoxPro 6 (Hentzenwerke, 1998) and Hacker's Guide to Visual FoxPro 3 (Addison-Wesley, 1996), both co-authored by Ted Roche. Several items were suggested by members of the FoxGang. Thanks to all those who made suggestions.

When is Equality Not Equality?

Picture this. You save a character string to two variables (say OldValue and NewValue). Then, you bind NewValue to a textbox and let the user loose on it. When she's done typing (perhaps in the Valid method), you compare the two values with a command like:

```
IF NewValue=OldValue
   * no change
ELSE
   * value has changed, do something
ENDIF
```

Being a sensible programmer, you decide to test this code before you actually let the users have it. You run the form, and add some more data (probably "test" or "asdf") at the end of the textbox. You've made a change, something should happen. But it doesn't. Huh?

Since the dawn of Xbase, new users have cracked their heads against this item. In a comparison, "=" doesn't necessarily mean "is equal to." Sometimes, it really means "starts with." To complicate things further, the rules are different in Xbase code than in SQL code.

The original design here was well intentioned. (But we all know where good intentions lead.) When you search for a record in a table, you may not know all the information to identify that record. Maybe you only remember that the name of the customer on the phone begins "Sch". You don't want to admit that you've forgotten his name. Xbase lets you search anyway in this situation and, depending on the command you use, finds either the first or all records matching the specified criteria. If you issue:

```
BROWSE FOR LastName="Sch"
```


the browse contains all the Schwartzes and Schmidts and Schnitzelwoofers it can find. This is convenient. But, the same behavior can be a real problem elsewhere.

Let's look at all the variations here. Start with Xbase behavior. In this case, whether "=" means "is equal to" or "starts with" depends on the setting of SET EXACT. With the default setting of OFF, you get the behavior described above. The comparison ends when the string on the right runs out. If the strings are identical to that point, the comparison returns .T. If the right-hand string is longer than the left-hand string, the comparison returns .F., even if the strings match to the end of the left-hand string. So:

```
"Schmidt"="Sch"
```

is .T., while:

```
"Sch" = "Schmidt"
```

is .F.

In some cases, just turning a comparison around is enough to solve the problem. But you have to know which string is shorter for that to work. There must be a better way. A rule of thumb in FoxPro is that there's always more than one way to do something - the trick is finding the best way. In fact, there are two ways, with one of those better than the other.

With SET EXACT ON, "=" really means "is equal to, except for trailing blanks". Trailing blanks are removed, and the strings are compared, character for character. So, with SET EXACT ON:

```
"Schmidt"="Sch"
```

returns .F. but

```
"Schmidt"="Schmidt"
```

returns .T., as do:

```
"Schmidt    "="Schmidt"
```

and

```
"Schmidt"="Schmidt  "
```

That's one way, but it has two flaws. First, it uses a SET command. Since the SET command can appear anywhere before the comparison, it's too easy to lose track of what the current setting is. Second, changing SET EXACT means we can't get the desirable search behavior described above. SET EXACT is a global setting and we're using it to handle a local situation.

The other method for performing an exact comparison is to use "==" (two equal signs) instead of "=". Read "==" as "exactly equals" (what you probably thought "=" meant in the first place). The "==" operator is extremely demanding. The strings must be the same length and match character for character for the comparison to be true. So:

```
"Schmidt"=="Sch"
```

is, of course, .F. Equally logically,

```
"Schmidt"=="Schmidt"
```


returns .T. But:

```
"Schmidt    "=="Schmidt"
```

and

```
"Schmidt"=="Schmidt     "
```

return .F. (If you need to use == when there may be trailing blanks, use TRIM() or ALLTRIM() before the comparison.)

So far, so good. Now what about SQL code? As in Xbase code, "=" doesn't necessarily mean "is equal to". However, in a SQL query, it doesn't matter which side of the equal sign is shorter. The comparison ends at the end of the shorter string. If the strings match to that point, the comparison returns .T.

SET EXACT has no effect on SQL code. Instead, SET ANSI serves the same purpose. With ANSI OFF, "=" means "one of these strings starts with the other". (Can you hear Ernie, Big Bird and the whole gang singing that?) With ANSI ON, "=" means "is equal to except for trailing blanks".

"==" isn't as strict in a query as it is elsewhere. Regardless of SET ANSI, "==" ignores trailing blanks in a query.

It's best to operate with SET EXACT OFF and SET ANSI OFF, using "==" when necessary. For those rare occasions where you need SET EXACT ON, change it right before the operation and change it back right after.

If you're an experienced programmer, the first example in this section (comparing NewValue to OldValue) probably looked a little phony to you. If the two strings are equal, you don't need to do anything. Only the case where they don't match matters. You're more likely to write:

```
IF NewValue<>OldValue   && or NewValue!=OldValue
                                                  && or NewValue # OldValue
                                                  && See? More than one way, again!
   * do something
ENDIF
```

This version is also affected by SET EXACT. To see how it works, you have to turn it sort of inside out. First the two strings are checked to see if they're equal (or what passes for equal with SET EXACT OFF). Then, the result is reversed. So, with SET EXACT OFF:

```
"Schmidt"<>"Sch"
```

returns .F. With SET EXACT ON, that comparison returns .T.

But I just said not to use SET EXACT ON. How can you check inequality without it? The trick is to do manually what FoxPro's doing automatically. Do the equality comparison first, then reverse it. To check for "exact inequality", use:

```
IF !(NewValue==OldValue)
```


Please REPLACE me - let me go

You have a routine that works with several tables to update some data. It seems to be working just fine until one day, the data doesn't get updated. You open up the Debugger and walk through the process. You can see that the field you're copying from has the right value and that the REPLACE command gets executed, but the target field doesn't change. What gives?

You've run into the dreaded "REPLACE at EOF()" problem. REPLACE, like many of the original Xbase commands, is a scoped command. Scope is a set of four choices (ALL, REST, NEXT n, RECORD n) that determine what records are affected by a command.

Each scoped command has a default scope. For non-destructive commands (like LIST, DISPLAY and BROWSE), the default is ALL. For destructive commands (like DELETE and REPLACE), the default is NEXT 1 to keep you from blowing away your data by accident.

What does this have to do with the update that's failing? You're using REPLACE and you only want to change one record. What's scope got to do with it? Scope applies to the current workarea, which may not be the target of the REPLACE (in fact, surely isn't if you're having this problem). If the record pointer in the current workarea is at EOF (end-of-file), no REPLACE occurs, because there's no field to be replaced. For example:

```
SELECT TableB
REPLACE TableA.Field1 WITH Field2
```

fails if TableB is at end-of-file.

When folks encounter this behavior for the first time, they usually holler "Bug! I found a bug!," but they're wrong. It's designed to work this way and it makes sense. If REPLACE could only update a single record and field at a time, this behavior wouldn't be necessary. But you can write a REPLACE like this:

```
REPLACE TableA.Field1 WITH Value1, ;
        TableB.Field2 WITH Value2, ;
        TableC.Field3 WITH Value3
```

Whose scope should apply in this case? The rule is always the same - the current workarea. If you didn't have that rule, what would you do if one of the destination tables was at end-of-file - replace some and not the others? The rule means you don't do any of them, which feels right.

The situation can be even uglier. Suppose the REPLACE above had a NEXT 5 clause as well. What happens if you run out of records in one of the destination tables before you've processed 5 records? Do you go on with the rest? The rule as it stands is simple - if there are enough records in the current workarea, the REPLACE goes through. If you reach end-of-file in the active workarea, you stop replacing.

The original designers probably shouldn't have let you write a REPLACE like the one above - then, this whole issue would be moot. But they did and until Visual FoxPro introduced the concept of transactions, there was no way to deal with an incomplete process, so there needed to be a rule that worked in all situations. None of which keeps this from being

terribly confusing.

The solution to this problem is simple. Use the IN clause of REPLACE to tell FoxPro to switch workareas on the fly. So, the first example in this section is better written as:

```
REPLACE Field1 WITH TableB.Field2 IN TableA
```

The multi-table replace above is better written as a series of REPLACEs, each one either SELECTing the right table first or using IN to choose the right work area. Wrap the whole thing in a transaction if it's appropriate.

Perform an APPENDectomy at Home

You have a table in which a number of records have been marked as ready for archiving by setting a logical flag. You want to add those records to your archive file prior to deleting them. The archive table has a subset of the fields in the main table and doesn't include the archive flag. You issue commands something like this:

```
USE archive
APPEND FROM working FOR lArchive
```

Boom, "Variable 'LARCHIVE' is not found." Oh, right, you need to use an alias for lArchive since it's not in the current work area. You change it to:

```
USE archive
APPEND FROM working FOR working.lArchive
```

Nope, this time you get "Alias 'WORKING' is not found. That's right, the Working table isn't open. Let's try it one more time.

```
USE Working
SELECT 0
USE Archive
APPEND FROM working FOR working.lArchive
```

You take a look at Archive. Still didn't work - you have either all the records in Working or none of them. What's going on here?

The mechanics of APPEND FROM with a FOR clause have been baffling FoxPro users for years. It's another case where the history of a command leads it to behave in an incredibly counterintuitive way.

The secret is that APPEND FROM evaluates the FOR expression as if the record had already been copied into the source table. I suspect this process takes advantage of the "phantom record" at end-of-file by putting the record to be added there. Then it evaluates the condition. If the condition is true in the new table, the record is kept and a new phantom record added. If the condition is false, the record is discarded.

In the example, since lArchive doesn't exist in the destination table, the flag isn't copied with the record. So, in the first two attempts, FoxPro couldn't find lArchive to evaluate it. The last case is more subtle. Working.lArchive was found, but only the value for the current record (in this case, the first record in the table) was evaluated - for every single record. If lArchive was .T. for the first record, all the records got copied; if it was .F., none were copied.

Why does APPEND FROM work this way? Two reasons, pretty closely related.

First, APPEND FROM can read data from a variety of file formats. The optional TYPE clause lets you specify what kind of file you're reading. File types supported include delimited files, SDF (System Data Format) files, and an assortment of files created by other applications such as Lotus 1-2-3, Excel, Paradox and others. FoxPro knows enough about those formats to be able to parse the data and plop it into fields, but not enough to evaluate the conditions before doing that. Second, in older Xbase versions, the source table wasn't allowed to be open in another work area. As with the foreign formats, the data wasn't available in the right form to do comparisons.

So now you know how it works and why it works that way. So what? What you really want to know is how to get the results you want. There's no one step way to do this. There are several methods involving two steps. In each case, you copy the data from the original table into a temporary table, then append it from there to the ultimate destination. One method is:

```
* The NOFILTER forces FP to create a real cursor,
* which is needed for this approach to work.
SELECT * FROM working ;
  WHERE lArchive;
  INTO CURSOR holding NOFILTER
SELECT archive
APPEND FROM DBF('holding')
USE IN holding
```

In versions of FoxPro before VFP, the delete flag of a record wasn't copied by APPEND FROM. So, this behavior applied to commands like APPEND FROM working FOR DELETED() as well. Beginning in VFP 3.0, the delete flag is copied with the record.

What kind of a DELI is this?

APPEND FROM and COPY TO have another strange behavior as well. One of their options lets you append "delimited" data, that is, data in a particular format involving delimiters between data items. But the syntax and terminology for APPEND FROM <file> TYPE DELIMITED is a mess.

The real problem is that delimited data actually involves both delimiters and separators. A delimiter is a character used at both ends of an item to indicate the boundaries of the item. For example, the quotes around the word "spelunking" in this sentence are delimiters. Not surprisingly, a separator separates consecutive data items. For example, the commas in the list "north, south, east, west" are separators.

DELIMITED data usually has both delimiters and separators. The default delimited format has quotes as string delimiters (other data items don't have delimiters) and commas as separators. The following is an example of the default delimited format:

```
"Smith","Fred",37,12/27/1974
```

(Incidentally, note that it's sensitive to the DATE and CENTURY settings.)

However, APPEND FROM and COPY TO allow some variations on the format. Each accepts a WITH clause after TYPE DELIMITED that specifies a character. You can put any

character you want or you can specify the keywords BLANK or TAB. Each of these options gives you different results.

TYPE DELI WITH TAB uses a tab as a separator, not a delimiter and uses the default quotes as a delimiter. The data above looks like this when we use COPY TO ... TYPE DELI WITH TAB:

```
"Smith"  "Fred"   37    12/27/1974
```

TYPE DELI WITH BLANK takes a different approach. It omits delimiters entirely and uses spaces for separators. In this format, the data looks like:

```
Smith Fred 37 12/27/1974
```

Finally, when you specify a character of your choosing, you get results that are consistent with the definitions of separator and delimiter. The items are delimited by the character specified and separated by commas. In this case, you have no control over the separator. Here's the same data again, using COPY TO ... TYPE DELI WITH ~:

```
~Smith~,~Fred~,37,12/27/1994
```

The DELIMITED WITH CHARACTER clause added in VFP 5 gives you control over the separator, as well. For example, if COPY TO … TYPE DELI WITH CHAR ! gives you:

```
"Smith"!"Fred"!37!12/27/1994
```

with the default delimiter and the separator you specify.

You can combine the two clauses to specify both a delimiter and a separator. For example, COPY TO … TYPE DELI WITH ~ WITH CHAR ! gives the following results:

```
~Smith~!~Fred~!37!12/27/1994
```

Most of the time, you can export data from another source in the default delimited format (with quotes as delimiters and commas as separators). When you can't do that, the two DELIMITED clauses give you a pretty good shot at reading the file in whatever format the other application can generate. If you can't read it by specifying both a delimiter and a separator, you'll have to either manipulate the output file (try using the FileToStr(), StrTran() and StrToFile() functions, if the file isn't huge) or use the low-level file functions to read the data in.

The "F" really does mean "Format"

Another TYPE option for APPEND FROM and COPY TO is SDF, which stands for "System Data Format". This means that each field has a fixed length and each record ends with a carriage return/line feed pair. This can be a very handy, read-able format, but you need to be careful with dates.

The designated format for dates in SDF files is YYYYMMDD and that's what COPY TO generates. If you APPEND FROM an SDF file using that format, all is well. But, if your dates are in another format, like MM/DD/YYYY, watch out. Your chances of getting the right data into the table are just about nil; if it happens, it's only through an accident of the data itself.

6.0/SP3 indicates that FoxPro has always expected the YYYYMMDD format. However, its reaction to dates that, instead, use the current DATE setting has varied over the years. Only FoxPro 2.0 gets it right – it rejects such dates without question and leaves the field empty. Other versions import the date and get the year wrong. The exact details of the wrongness vary from version to version. VFP 6 uses the century portion of the year as the last two digits and puts "00" in front. Some versions use "19" for the century portion. In some cases, if the date to be input has only a two-digit year, FoxPro gets it right (though it places all such dates in the range 1900-1999).

It's reasonable for FoxPro to expect the specified format, so failing to work on anything but YYYYMMDD is not a bug. However, it would be much better if it just rejected any dates not in that format rather than accepting and mangling them. It worked in FoxPro 2; it's a shame they broke it later.

You're just my TYPE() or is that my VarType()?

The TYPE() and VarType() functions are both handy for building black-box code, but each has a trap that's easy to walk into.

Suppose you've just discovered these two functions and you start testing them in the Command Window. You start with:

```
?TYPE('a string')
```

and FoxPro responds

```
U
```

"U" stands for undefined. That's odd. What about VarType():

```
?VarType('a string')
```

The response is:

```
C
```

That's better, but what about TYPE()? Maybe it doesn't like constants. How about a variable?

```
x="A string"
?TYPE(x)
```

you still get

```
U
```

On the other hand:

```
x="A string"
?VarType(x)
```

returns:

```
C
```

as expected. What's going on here?

The way TYPE() works is confusing at first glance (and at later glances, too, actually).

VarType() is a little more straightforward, but does have its own bumps in the road.

TYPE()'s main purpose in life is to help you build black-box code. When you're working in that situation, you often have a variable that holds the name of something. For example, it's not unusual to pass the name of a field to a function or to pass the name of an array rather than the array itself. TYPE() is set up to assume you're in that kind of situation and, therefore, need an extra level of indirection.

TYPE() first evaluates the character string you pass it. Then, it evaluates (or pseudo-evaluates) the result and gives you back the type of the result. Let's run through that again with an example. Suppose you have a character variable cField containing the name of a field you want to operate on, say, Birthdate. Here's what happens when you issue:

```
?TYPE(cField)
```

FoxPro evaluates cField and gets back:

```
"Birthdate"
```

Then, FoxPro looks at the contents of the string (Birthdate without the quotes) and tells you what type that is. In this case, you'd probably see:

```
D
```

So far, so good. But what if you don't have a variable containing the name of the thing you want the type of. Do you have to create one? Nope, the trick is to surround the expression with quotes which TYPE() can then remove. For example:

```
?TYPE('3+2')
```

returns

```
N
?TYPE('LastName="Smith"')
```

returns

```
L
```

(Incidentally, EVAL() uses the same technique on its parameter. The difference is that it returns the result of evaluating the contained expression rather than its type.)

If TYPE() does the job, why do we need VarType() and how is it different? Unlike TYPE(), VarType() uses only one level of indirection. It evaluates what you pass it (actually, VFP usually evaluates it before calling VarType(), which leads to the problem described below) and returns the type of the result. So, with cField set to "Birthdate" as above, if you make this call:

```
?VarType(cField)
```

VFP evaluates cField and finds "Birthdate". Since that's a character value, VarType() returns "C". It never looks at Birthdate as a name.

Why bother? VarType() is faster than TYPE(). My tests show it as anywhere from two to six times faster.

So you should use only VarType() from here on out, right? No. TYPE() works in some cases

where VarType() doesn't. Although VarType() does return "U" if you pass it a variable that doesn't exist (as in VarType(NonExistent)), it doesn't have the ability to dig through containership hierarchies to determine whether a property exists. So, while you can successfully use code like:

```
IF TYPE("oForm.Name") = "U"
```

the corresponding use of VarType:

```
IF VarType(oForm.Name) = "U"
```

gives an error message if oForm doesn't exist or is null. (The exact message depends on the circumstances.) That's because VFP attempts to evaluate the parameter before passing it to the function. Sort of. Actually, VFP attempts to get to the last level of the containership hierarchy, then is willing to deal with a non-existent property. So, if oForm exists, but you ask about a property that doesn't (say, VarType(oForm.Fred)), VFP returns "U".

Given these complications, maybe you should use TYPE() all the time. Not that, either. The trick is to know what VarType() was designed for and use it for that.

There are two main reasons to use VarType(). The first is to check parameters. By definition, when you enter a method or procedure, variables are created for all parameters. So, parameter variables always exist.

The second place for VarType() is in checking whether an object exists. The traditional code to so is:

```
IF TYPE("oObject") = "O" AND NOT ISNULL(oObject)
```

Since VarType() returns "X" for a null object, you can reduce that to:

```
IF VarType(oObject) = "O"
```

For digging into complex object hierarchies, stick with TYPE().

Support GROUP

The GROUP BY clause of SELECT-SQL is pretty neat. It lets you take a bunch of records and consolidate them into a single record, counting them or computing totals or averages along the way. Just the way to figure out how many customers you have in each state or the average temperature each month of the year.

But GROUP BY has a hidden behavior. Take a look at this query (using sample data that comes with Visual FoxPro):

```
SELECT City, Country, COUNT(*) ;
   FROM TasTrade!Employee ;
   GROUP BY Country
```

You'll find out there are 7 employees in the US, 5 in the UK and 3 in France, but what does the City field in each row of the result mean? Is that where those employees live? All of them?

In fact, it's where one employee from that country lives. Do we know which employee? The

first one listed? The last? The first one alphabetically? The last? In fact, it is the last one in record order.

Okay, that's not unreasonable, but now look at this query:

```
SELECT Last_Name, Country, MIN(Birth_Date) ;
   FROM TasTrade!Employee ;
   GROUP BY Country
```

Here the idea is to find the oldest employee in each country. But whose last name gets put in the record? Again, it's the one from the last record for that group in the original data. There's no connection between the last name listed and the person whose birth date is shown.

What's going on here? GROUP BY is meant to consolidate and aggregate data and the query doesn't quite know what to do with a field that is neither a part of the GROUP BY clause nor an aggregate function. So, the query simply takes the last value it finds. (The actual value chosen is random, so other versions or languages may pick a different value from the set.) When you use MIN() and MAX(), there's no relationship between the minimum or maximum value found and the values in any fields not part of the GROUP BY clause. The rule of thumb is that every field in a query with GROUP BY should either be used in the GROUP BY clause or be part of an aggregate function.

How do you get the last name of the oldest employee in each country? Try it like this:

```
SELECT Last_Name,Country,Birth_Date ;
   FROM TasTrade!Employee ;
   WHERE Country+DTOS(Birth_Date) IN ;
      (SELECT Country+DTOS(MIN(Birth_Date)) ;
          FROM TasTrade!Employee Emp2 ;
          GROUP BY Country)
```

This query first finds the minimum birth date from each country, then pulls out the records having that birth date. If two people in the country have the same birth date and it's the oldest, you'll get two records for that country in the result.

If you think that query is too hard to maintain (as I do), break it into two steps:

```
SELECT Country, MIN(Birth_Date) AS MinBirthDate ;
   FROM TasTrade!Employee ;
   GROUP BY Country ;
   INTO CURSOR MinDate

SELECT Last_Name, Country, Birth_Date ;
   FROM TasTrade!Employee ;
     JOIN MinDate ;
       ON Employee.Country = MinDate.Country ;
      AND Employee.Birth_Date = MinDate.MinBirthDate
```

But It Used to Work

When converting code to VFP5 or VFP6, you may find that a lot of code that worked just fine in VFP3 or earlier versions of FoxPro is turning up with syntax errors. You're not going crazy - it's true. There was a major change to the parser in VFP5 that's likely to turn up a lot

of old errors.

Way back in Xbase history days, someone decided that you should be able to put comments on the same line as the structured programming commands without having to use the comment indicator. That is, you can write:

```
DO WHILE .NOT. EOF()
   * process records
ENDDO WHILE .NOT. EOF()
```

The rule applies to the components of the branching and looping commands (IF, DO CASE, DO WHILE, FOR and SCAN). Until VFP5, however, you could even put such comments on lines of those commands that contained expressions. For example, code like this was not rejected by the compiler:

```
IF dBirthDate>GOMONTH(DATE(),-120)   At least 10 years old
   * do something
ENDIF
```

It looks real nice, but it caused a rather nasty, subtle problem. When FoxPro parsed one of these lines, it stopped as soon as it reached something syntactically incorrect. That's right, as soon as the parser found a syntax error on an IF or DO WHILE or CASE line, it figured it had a comment and it gives up. Here's an example:

```
IF x<3 .AND y>7
```

As far as all versions of FoxPro prior to VFP 5 were concerned, that line only checks whether x is less than 3. Those versions totally ignored the second part of the condition. (Back in the bad old days, dots were required around the Boolean logical operators .AND., .NOT. and .OR.)

I've heard of people finding a mistake like this after an application's been running for years and years when someone finally notices weird behavior. Frankly, this was a terrible design decision on the part of the original developers. I'm delighted that they've finally tightened up the rules.

In VFP 5 and 6, inline comments without a comment indicator are permitted only on those lines that don't contain anything else that needs to be parsed and evaluated. The addition of syntax coloring, however, provides a good reason not to use them there either.

The Single Letter Blues

On the whole, the designers of Visual FoxPro have done a tremendous job marrying object-orientation to Xbase. But there are places where the marriage seems a little rocky. The use of single letter identifiers is one of them.

Traditionally, the letters A through J are alternate names for the first 10 work areas. (When the number of work areas went up to 25, Fox Software didn't extend this convention. It's just as well - they'd have a terrible time finding 32,767 different characters to represent the work areas in Visual FoxPro – we'd have to go to Unicode.) In addition, the letter M was reserved to indicate that what followed was a memory variable.

None of this ever caused anyone any trouble because when the letters were used for work areas, they were always followed by either the pseudo-arrow notation "->" or a dot "." to indicate that what followed was a field name.

Enter OOP. VFP's Object-oriented extensions use the same dot to spell out the complete name of an object, like frmMyForm.grdMainGrid.colName.txtName. The dot lets you walk up and down the containership hierarchy.

So what's the problem? There is none, unless you try to use one of the letters A-J or M for the name of an object. For backward compatibility reasons, you can't do that. Code like the following:

```
a = CREATEOBJECT("form")
a.caption = "My Form"
```

is doomed to failure. There is a work-around - prefix the variable name with m. as in:

```
m.a.caption = "My Form"
```

Of course, single letter variable names are a lousy idea anyway (most of the time - x, y, z, and q are still handy for quick and dirty testing), so the other work-around for this isn't terribly painful. It's like the old joke: "Doctor, it hurts when I do this." "So don't do that." Use names longer than one character and you'll never run into this problem.

What is Reality, Anyway?

One of FoxPro's optimization tricks for queries is that it doesn't create a real cursor ("real cursor" - isn't that an oxymoron?) unless it has to. Whenever possible, FoxPro simply filters the original data.

When is this possible? When the query involves only a single table, is fully optimized, and there's no post-processing (like ORDER BY or GROUP BY). For example:

```
SELECT Last_Name, First_Name ;
   FROM TasTrade!Employee ;
   INTO CURSOR Names
```

The problem is that certain operations fail and others misbehave when the cursor is really just a filtered view of the original data. For example, you can't put such a cursor in the FROM list of another query. You also run into trouble if you APPEND FROM such a cursor.

VFP 5 and VFP 6 offer an easy solution to the problem. The NOFILTER clause (which can be used only with queries INTO CURSOR) indicates that VFP should create a real cursor, even if filtering the original is possible. So, if you need to do something with the result of the query that is badly affected by filtering the source, add NOFILTER. For example:

```
SELECT Last_Name, First_Name ;
   FROM TasTrade!Employee ;
   INTO CURSOR Names NOFILTER
```

Unfortunately, of course, it does take longer to create a real cursor than to filter the raw data. So, only force the issue when you need to.

Whaddaya Mean, They're Not the Same?

There are two methods provided for filling in the contents of a list or combo box manually. Their names are similar and you'd think they do pretty much the same thing. But there's a subtle difference between them that leads a lot of people to think they've found a bug.

The methods are AddItem and AddListItem and, in fact, there are two differences between them. The first difference is paralleled throughout the properties and methods that handle lists and combos.

There are two ways to look at the collection of items in a list or combo - the order in which they were added or the order in which they now appear. The order in which they were added is ItemId order - each item is assigned a unique sequential id number when it's added to the list. (Actually, it doesn't have to be sequential - you can specify the ids if you want.) The id never changes, no matter what you do to the item. Until that item is removed, it has that id.

Items also have an Index that is the item's current position in the list. The index is highly fluid. It changes when other items are added or removed. It changes when the list gets sorted. It changes when the user moves items around with the mover bars.

Almost every property and method for handling lists and combos is double - there's an Index version and an ItemId version. For example, the currently selected item is referenced both with ListIndex and ListItemId. The list of items can be found in both List (in Index order) and ListItem (in ItemId order). (In fact, the List and ListItem collections are really two views of the same underlying data - changes to one change the other.)

So this is the expected difference between AddItem and AddListItem. AddItem operates with Indexes; AddListItem operates with ItemIds. But there's another, far trickier difference.

AddItem always adds a new item to the list. No matter what parameters you pass it, the result is that ListCount increases. AddListItem, on the other hand, may or may not add a new item to the list. If you give it the ItemId for an item that already exists, it simply changes the contents of that item - it doesn't add a new item.

This subtle difference leads to a more visible and confusing difference. In a list or combo with multiple columns, AddItem is incapable of populating all the columns. The first time you call it with a particular index, it adds a new item at that position. But, if you call it again with the same index, but a different column (hoping to fill in that column of the newly-added item), it again adds a new item, pushing the old one down. The result is far more items than you expected with data in only one column each. (A bug in VFP 5 and later actually puts data in the specified column *and* the first column when AddItem is called with a column parameter.)

You have to either use AddListItem to populate multiple columns or use AddItem for the first column, then put the data for the other columns right into the List collection property. In either case, the NewItemId property, which contains the ItemId for the most recently added item is helpful. Here are two ways to fill a multi-column list. In both cases, we're putting first

name, last name and title from the TasTrade Employee table in the list.

```
* Fill the list using AddListItem. We don't care
* what ItemId is assigned.
SELECT Employee
SCAN
   THIS.AddListItem(First_Name)
   THIS.AddListItem(Last_Name, THIS.NewItemId, 2)
   THIS.AddListItem(Title, THIS.NewItemId, 3)
ENDSCAN

* Fill the list by addressing it directly. Again, we don't care
* what ItemId is assigned.
SELECT Employee
SCAN
  THIS.AddItem(First_Name)
  THIS.ListItem(THIS.NewItemId, 2) = Last_Name
  THIS.ListItem(THIS.NewItemId, 3) = Title
ENDSCAN
```

So Who Needs Parents Anyway?

The object-oriented additions to Visual FoxPro include two different hierarchies: the inheritance hierarchy and the containership hierarchy. It's easy to get the two confused, especially since they use some similar terminology in VFP.

The inheritance hierarchy refers to the chain of class definitions on which an object is based. All objects in VFP ultimately derive from the base classes. But you can define your own classes (subclasses) based on those base classes. And then, you can define subclasses of your own classes and so on and so forth, for as many levels as you want.

What makes subclassing so attractive is that each subclass inherits all the characteristics (properties) and behaviors (methods) of its ancestors. You can then make changes in the subclass to get what you need. But anything you don't change comes down the hierarchy. When you call a method of an object, VFP looks at the object's method. If there's code in that method, it gets executed. If not, VFP goes to the object's class and checks for code in the same method there. Again, if there's code there, it's executed. If not, we go to the class that subclass is based on and look there. If no code is found, we go up another level. And so on and so forth until we either find some code or reach the VFP base class.

In OOP terminology, the class one level up the hierarchy is called the *superclass*. (Makes sense. "Sub" is Latin for "under," while "super" means "over.") Unfortunately, VFP doesn't use OOP terminology for this and instead refers to this class as the *ParentClass*. Each class has the ParentClass property that contains the name of the class from which this class was subclassed.

So far, so good. Now what about the containership hierarchy? Some of the base classes in VFP are capable of holding other objects. For example, a formset holds forms, a form holds all kinds of controls, a grid holds columns, which in turn hold headers and other controls.

The chain of who's inside of whom is the containership hierarchy. You use it to construct complete references to contained objects. For example, a textbox on a page of a

pageframe inside a form might be called:

```
MyForm.pgfData.pagName.txtLastName
```

The object that contains another object is called the contained object's parent (and can be referenced that way from inside the contained object - THIS.Parent is a reference to the container). And that's why things are confusing. The containership hierarchy has parents; the inheritance hierarchy has parentclasses.

And don't forget that we use the term "parent" for the one table in a one-to-many relationship and for a window that contains other windows. Nobody needs this many parents.

Push 'em Back, Shove 'em Back

Visual FoxPro provides both buffering and transactions to help you deal with conflicts and system problems when storing data. Buffering lets you work on a copy of the data, then either commit your changes or restore the original. Transactions let you commit changed records as a group, bailing out if you need to and ensuring that either all the changes go through or all of them don't. Both techniques are useful and, best of all, they work together. You can buffer data while you work on it, and then, when you're ready to commit the changes, wrap the whole thing in a transaction so it's all or nothing.

But when you work with buffering and transactions together, there's one behavior that doesn't make sense at first glance. When you rollback a transaction after issuing TableUpdate() for some of the tables involved, then examine the buffered tables, the changes still appear in the records involved. What's going on here?

Let's start by taking a look at what's really going on when you use buffering and transactions. When you buffer a table, VFP makes two copies behind the scenes. One copy is kept intact and is accessible via the OLDVAL() function. The other copy is what you're working on when you think you're touching the table - you use the Alias.Field syntax - call this the working copy.

When you issue TableUpdate(), the values in the working copy are copied to the disk and the OLDVAL() copy is updated. When you issue TableRevert(), the current data on the disk is copied to both the working copy and the OLDVAL() copy (except with remote views, where the OLDVAL() values are copied to the working copy - use Refresh() to update the data from disk).

Transactions add another level of buffering to the whole process. When you're in a transaction and issue TableUpdate(), the data from the working copy is copied to a transaction buffer. When you reach END TRANSACTION, the transaction buffers are copied to the disk. (Actually, with nested transactions, it can be more complicated than that, but that's not relevant to the problem at hand.) So far, so good.

But what happens if you issue ROLLBACK to kill the transaction? The transaction buffers are discarded, but the changes are still sitting in the working copy. You have to issue TableRevert() to throw them away or try again to commit the whole thing with

TableUpdate(). In other words, when you issue TableUpdate() within a transaction, it's not a permanent change as it is without the transaction - the update doesn't become permanent until the transaction is committed and TableRevert() is still possible.

But Why Doesn't It Run?

Here's a behavior that drove people nuts in FoxPro 2.x and, even though the whole context surrounding it has changed in Visual FoxPro, continues to drive people nuts.

Say your user is entering a new record and is sitting on a field that requires validation. After entering bad data for that field, but before moving focus to another field, the user clicks on the Save button on your toolbar. Or chooses Save from the menu. Whoosh - the data is saved, including the bad data. What happened?

In VFP, as in FoxPro 2.x, the Valid routine for a text field doesn't execute until focus leaves that field. Choosing a button on a toolbar or picking a menu item doesn't change focus and therefore doesn't fire the Valid method or the LostFocus method, of course. (In fact, a toolbar never has focus, which lets you do pretty cool things.)

Why does it behave this way? Because we want it to. It feels wrong in this situation, but suppose the toolbar or menu item the user chose was Select All or Paste. We sure wouldn't want the focus to change (and the data to be committed and the Valid and LostFocus methods to fire) in that case.

So how do we make sure the data gets validated? Simple - make sure focus changes. One way to do this is to reset focus to the same field:

```
_SCREEN.ActiveForm.ActiveControl.SetFocus()
```

This simple version works unless grids are involved. When a control in a grid has focus, _SCREEN.ActiveForm.ActiveControl points to the grid itself, not the control. In this case, a much more convoluted piece of code is needed. Thanks to Drew Speedie, here's the code you need:

```
IF _SCREEN.ActiveForm.ActiveControl.BaseClass<>"Grid"
   _SCREEN.ActiveForm.ActiveControl.SetFocus()
ELSE
   * Need to drill down
   LOCAL oActiveControl, cActiveControl, oActiveColumn, cRefName
   oActiveControl = _SCREEN.ActiveForm.ActiveControl
   * Get Name of active control in grid
   cActiveControl =
   oActiveControl.Columns[oActiveControl.ActiveControl].CurrentControl
   * Get reference to column
   oActiveColumn = oActiveControl.Columns[oActiveControl.ActiveControl]
   * Create name to allow reference
   cRefName = "oActiveColumn."+cActiveControl
   oActiveControl = &cRefName
   oActiveControl.SetFocus()
ENDIF
```

Since menus and toolbars don't get the focus, _SCREEN.ActiveForm is still the same form as before. We just set the focus back to the object which had it, which triggers that control's

Valid, LostFocus, When and GotFocus methods.

Why Do I Need It? It Doesn't DO Anything

Visual FoxPro 3 was an Automation client, which allowed us to send directions to Word or Excel or any other Automation server. Visual FoxPro 5 and 6 are Automation servers as well and allow us to create our own custom Automation servers. These are programs that can be started from other applications and manipulated by those applications.

It's incredibly easy to create a custom server. Just define a class and give it the OLE Public characteristic. In code, it's as simple as adding OLEPUBLIC to the DEFINE CLASS command. In the Class Designer, you check OLE Public on the Class page of the Class Info dialog. In the Project Manager's Project Info dialog, you provide additional information about the server. When you build the project, each class defined as OLE Public is registered as a server. Easy, right?

Except for one thing. Say you've defined your server classes and stored them all in a project. Now you choose Build and then choose any of the options for building servers (the exact list varies by VFP version). Whoops, what's that error? "Cannot build without a main program."

But what would you put in the main program of a project whose purpose is simply to hold onto server classes? In fact, anything at all. All you have to do is create a PRG and set it as the main program for the project. Of course, you should put a comment in there explaining its reason for existence.

Join Conditions

VFP 5 and 6 include syntax for the SQL-SELECT command that allows you to specify the joins between tables in the FROM clause, using the JOIN and ON keywords. Here's an example:

```
FROM Customers ;
  JOIN Orders ;
    ON Customers.Customer_Id = Orders.Customer_Id
```

Along with the new syntax, VFP gained the ability to perform outer joins (queries where unmatched records appear in the result set). Four types of joins are now supported: inner joins (the type we've always had) and three kinds of outer joins that differ in which unmatched records appear in the result. To perform an outer join, just add the LEFT, RIGHT or FULL keyword before the JOIN keyword.

Whether you're performing inner joins or outer joins, what if you want to join more than two tables? It turns out there are two different ways to specify the joins and each is appropriate in different situations.

One approach is to nest the joins. That is, string out a series of tables, then list all the ON clauses with the actual join conditions. For example:

```
SELECT Customer.Company_Name, Product.English_Name ;
   FROM Customer ;
     JOIN Orders ;
       JOIN Order_Line_Items ;
         JOIN Products ;
         ON Order_Line_Items.Product_Id = Products.Product_Id ;
       ON Orders.Order_Id = Order_Line_Items.Order_Id ;
     ON Customer.Customer_Id = Orders.Customer_Id
```

The nesting shown is intentional because the ON clauses are matched with the table listings in reverse order. That is, the first ON clause is applied to the last pair of tables listed.

The order of the ONs matters. If you don't match the join conditions with the right set of tables, you can get some really weird results. The query above (which uses the TasTrade sample data) produces a list showing the company who ordered a product and the English name of the product (one record for each time anything was ordered). If we move the ON clauses around, the results change. For example, using this query instead:

```
SELECT Customer.Company_Name, Products.English_Name ;
   FROM Customer ;
     JOIN Orders ;
       JOIN Order_Line_Items ;
         JOIN Products ;
         ON Orders.Order_Id = Order_Line_Items.Order_Id ;
       ON Customer.Customer_Id = Orders.Customer_Id ;
     ON Order_Line_Items.Product_Id = Products.Product_Id
```

gives a listing in which each company appears as many times as it has records in Orders, but is always matched with a single product name. (Under some conditions, this query won't even run.) Other variations in the order of the ON clauses lead to other incorrect results.

So, the first gotcha with joins is that nested joins get the actual join conditions (the ON clauses) in reverse order of the table listing.

The other approach to join conditions is to list them sequentially, alternating JOIN clauses with ON clauses. This approach is handy when you want to join multiple child tables to a single parent.

For example, suppose you want to see which salespeople are sending orders by which shipper. You need to join each of the two tables to Orders, but there's no relationship between Employee and Shippers. You could use a nested join like this:

```
SELECT Employee.Last_Name, Shippers.Company_Name ;
   FROM Employee ;
     JOIN Orders ;
       JOIN Shippers ;
       ON Orders.Shipper_Id = Shippers.Shipper_Id ;
     ON Orders.Employee_Id=Employee.Employee_Id
```

but a sequential join like this one shows the relationships among the tables much more clearly.

```
SELECT Employee.Last_Name, Shippers.Company_Name ;
   FROM Orders ;
      JOIN Employee ;
        ON Orders.Employee_Id=Employee.Employee_Id ;
```


```
      JOIN Shippers ;
        ON Orders.Shipper_Id = Shippers.Shipper_Id
```

In this query, it's clear that you're dealing with two separate relationships between Orders and another table. The first query implies a parent-child-grandchild relationship, which isn't the case logically.

In this example, it doesn't matter which format you use because every order record has a matching shipper record and a matching employee record. However, in some situations, there's simply no way to get the desired results only by nesting. For example, suppose you want to show product information as well in the query above. Where do you add the joins for the Order_Line_Items and Products tables in the nested version? Clearly, Order_Line_items needs to be joined to Orders (and Products to Order_Line_Items), but if we do that, the Shippers table ends up joined to Products, like this:

```
SELECT Employee.Last_Name,Shippers.Company_Name, Products.English_Name ;
   FROM Employee ;
     JOIN Orders ;
       JOIN Order_Line_Items ;
         JOIN Products ;
           JOIN Shippers ;
           ON Orders.Shipper_Id = Shippers.Shipper_Id ;
         ON Order_Line_Items.Product_Id = Products.Product_Id ;
       ON Orders.Order_Id = Order_Line_Items.Order_Id ;
     ON Orders.Employee_Id=Employee.Employee_Id
```

When you run this query, one of two things happens. If the Shippers table is open, you get bad results - every record contains the same shipper information. If Shippers is not open, you get an error message that column Shipper_Id can't be found.

In fact, the hint that something is wrong with this query comes from the first join condition. Although the last two tables specified in the nesting are Products and Shippers, the join condition references Orders and Shippers.

Clearly, in this case, the nested approach won't do the trick. In fact, to get the results we want in this case requires a query that uses both nested and sequential joins. Here are two versions that work:

```
SELECT Employee.Last_Name,Shippers.Company_Name, Products.English_Name ;
   FROM Employee ;
     JOIN Orders ;
       JOIN Order_Line_Items ;
         JOIN Products ;
         ON Order_Line_Items.Product_Id = Products.Product_Id ;
       ON Orders.Order_Id = Order_Line_Items.Order_Id ;
     ON Orders.Employee_Id=Employee.Employee_Id ;
     JOIN Shippers ;
       ON Orders.Shipper_Id = Shippers.Shipper_Id

SELECT Employee.Last_Name,Shippers.Company_Name, Products.English_Name ;
   FROM Orders ;
     JOIN Order_Line_Items ;
       JOIN Products ;
         ON Order_Line_Items.Product_Id = Products.Product_Id ;
     ON Orders.Order_Id = Order_Line_Items.Order_Id ;
     JOIN Employee ;
```


```
  ON Orders.Employee_Id=Employee.Employee_Id ;
JOIN Shippers ;
  ON Orders.Shipper_Id = Shippers.Shipper_Id
```

In fact, there is a way to use only the nested syntax, but the resulting query is very hard to read and equally hard to maintain.

Worrying about the order of join conditions is a new problem, but given the added flexibility of all four types of joins, it's worth figuring out.

FLOCKs Aren't Just for Sheep

VFP 5 added two functions FoxPro programmers had long been asking for: IsFLocked() and IsRLocked(). Together with IsExclusive() and IsReadOnly(), these functions make it possible to determine the current lock status of a table or record without having to parse the results of SYS(2011). This means that we can write locking status functions that are language-independent. (SYS(2011) returns different values in different localized versions of FoxPro.)

However, there's a gotcha here. (Isn't there always?) The return values of these functions don't all interact. Specifically, IsRLocked() returns .T. only when the record was locked by an RLock() call. When a table was opened exclusively or locked with FLock(), IsRLocked() returns .F. for the records in that table. Here's an example:

```
SET EXCLUSIVE ON
USE States
? SYS(2011)       && returns "Exclusive"
? ISEXCLUSIVE()       && returns .T.
? ISFLOCKED()       && returns .T.
? ISRLOCKED()       && returns .F.
?
?"Now lock a record explicitly."
? RLOCK()       && returns .T.
? ISRLOCKED()       && returns .T.
```

This means that our lock status function has to check more than one possibility to come up with the right result.

A Private Execution

Another cool feature in VFP 5 and VFP 6 is the ability to highlight a section of code, right-click and execute it. It's much quicker and easier than copying to the clipboard, pasting into a test program and running that. It's also less prone to error since you'll make corrections right on the original rather than on the test copy (where you might forget to propagate them back to the source).

The Execute Selection option is available both in editing windows and the Command Window. However, when you use it in the Command Window, you may get some unexpected results.

One of the bonuses of testing in the Command Window is that all the variables you create in

the process are scoped public, so they stick around as long as you're playing with them. When you Execute Selection, even from the Command Window, VFP does behind the scenes what you used to do explicitly: the highlighted code is copied to a temporary file, then executed. Any variables created in the code are scoped to the temporary program, not made public, as they would be if you simply executed the line in the Command Window. For example, suppose you typed the following lines into the Command Window:

```
USE States
=AFIELDS(aFieldList)
```

The aFieldList array would then be available for examination and manipulation. Suppose, though, that you issued CLEAR ALL, CLOSE ALL to clean up, then went back and selected those two lines, and choose Execute Selection. When you try to check what's in aFieldList, the array is nowhere to be seen. It was scoped to the program that created it and, of course, destroyed when it went out of scope.

What's the solution? One solution is to create the variables you want before executing the code. In the example above, issuing DIMENSION aFieldList[1] does the trick. Most often, I run into this when I use CreateObject() or NewObject() to create an instance of a class from the Command Window, then manipulate that object to test it. Highlighting and executing the line that creates the object results in an object that's immediately destroyed. However, if I run the CreateObject()/NewObject() line by simply arrowing up to it and pressing Enter, I can then highlight and execute all the lines that operate on the object.

When is an Add not an Add?

So you've decided to create a custom class for something you need to do over and over. Great, that's the spirit. You open the Class Designer and get to work. You quickly realize that you need to add a custom property to keep track of some important value. No problem – you choose Class-New Property and up pops the New Class dialog (Figure 1 shows the VFP 6 version). You type in the name of your new property, carefully write a detailed description of its purpose, and click Close. You switch over to the property sheet, scroll down to the bottom to find your new property and, what's this? It's not there.

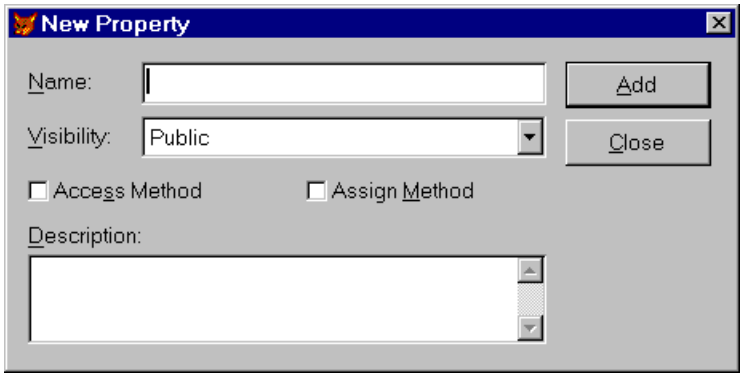

Figure 1 Adding a property – Easy enough to use, if you just know the trick.

This is a case of annoying interface design. You have to actually choose Add before the

property is added. Since Add is actually the default button in the dialog, that wouldn't be so bad except for one thing. Because the description goes into an editbox, it's not enough to press Enter when you're done the description to commit your new property. You have to either tab out of the editbox and press Enter, or remember to click Add before Close.

Interestingly, the Add Table or View dialog in the Data Environment has the same behavior, except that it seems to trip me up the opposite way. In that dialog (Figure 2), it's easy to add the same table or view twice when you really mean to close the dialog. Guess that just proves that the broader design of a using an Add button is somehow flawed (or that my expectations of how a dialog should behave are flawed).

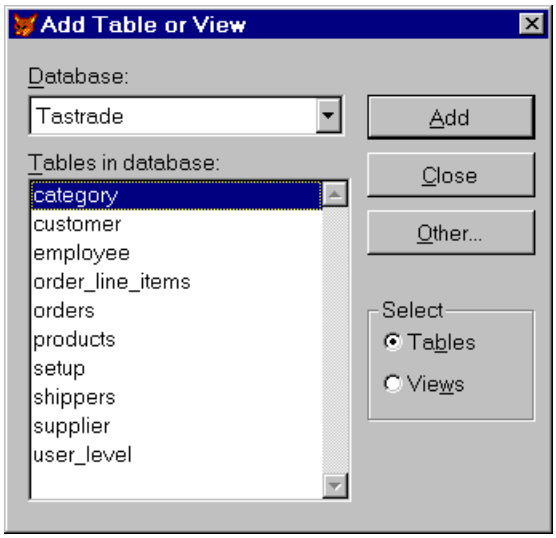

Figure 2 Adding to the Data Environment – Be careful you don't add the same thing twice.

It Ran Okay in VFP 5!

Normally, the folks at Microsoft (and, before them, the folks at Fox Software) go out of their way to make sure that, whatever they change in a new version of FoxPro, code that ran in the old version will still run in the new one. But in VFP 6, they broke that rule. Not only that, they did it on purpose. Say what?!

With the year 2000 nearly upon us, the Fox team figured it was about time to really make FoxPro developers aware of the bugs lurking in their old code. So they added the SET STRICTDATE command to help us find problems and to keep us from making them in the first place. When you turn STRICTDATE on, VFP lets you know if any of your code contains ambiguous dates. Great, that sounds good.

But here's the catch. By default, it's set to moderate strictness in the development environment. That means that any date or datetime constants that don't use the long unambiguous format (that is, {^1958-9-28} rather than {9/28/58}) cause an error.

Well, it's nice that they can find our errors for us, but why they heck didn't they make the old

way the default? Because if they had, we'd keep going along in our misguided belief that our old code was Y2K-compliant. They're forcing us to pay attention.

About now, you're probably wondering how you're supposed to upgrade your users to VFP 6 if all your dates are going to fail. That's easy. By default, STRICTDATE is set to 0 for them (that is, at runtime), but you'll want to set it to 2 on your development machine so you can catch and squash these bugs way before January 1, 2000.

Print What Where?

Letting users choose the printer for a report at runtime and having it automatically adjust itself for that printer's settings ought to be a piece of cake. Isn't that one of the things Windows is supposed to handle for us? Unfortunately, VFP tries to be too smart and ends up making things a lot harder than they need to be.

When you create a report with the Report Designer, information about the currently selected printer and its settings is stored in the first record of the report table (FRX). When you print the report, VFP checks that information and uses it. If the selected printer at design time was the Windows default, VFP is smart enough to use whatever printer the user chose. But if the selected printer was something else, VFP assumes that you meant it when you created the report with that printer chosen, and it expects to print to that printer.

Why does it behave this way? Most likely, the developers were trying to be helpful. They figured that, if you'd gone to the trouble of specially configuring a printer for the report, they ought to pay attention. The problem with this is that the people using your application probably don't even have the same printer you do. The other problem is that this is such a non-intuitive way of arranging for special settings that it's hard to imagine anyone doing it on purpose.

So how do you get VFP to print to the user's chosen printer and honor that printer's settings? Easy—throw out the stored settings. The information is in the Tag, Tag2 and Expr fields of the first record of the report. Just blank 'em out.

My friend Brad Schulz, who knows more about printing from FoxPro than anyone else, suggests making a copy of the report at runtime, blanking the Tag, Tag2 and Expr fields of the copy, then stuffing the Expr field (which is plain text in an INI-file type format) with the settings you really want. Then, use the copy to run the report.

That's All, Folks

Actually, that's not all. Given its long and varied history, Visual FoxPro has lots more things you might consider bugs when you first encounter them. It's always a good idea to try to consider the history and purpose of a behavior before you decide it's a bug. Sure, there are bugs in VFP, and some things whose design is questionable. But most of the language behaves pretty rationally once you understand what's going on.