Another Boring Article About Regular Expressions

Whil Hentzen

Fox developers have lived a long time without needing to know anything about regular expressions. But then that Linux thing became popular a decade ago, and suddenly literature all over the place started referencing them. Yet many developers, not seeing an immediate need, ignored them. “I've been using Fox since FoxBase+ any never needed them. I can't see why I'd need them now.” And, true, you don't NEED them. But regular expressions fills the need addressed in this plea “I don't need more input. I need to be able to create more output.” Regular expressions might be that tool that will allow you to fit 36 hours of work into a 24 hour day some time. So bear with me; in this article, I'll explain why you MIGHT want to have the ability to use regular expressions in your toolkit, and how to get started.

1. How can regular expressions be useful to me?

Regular expressions can save you time by providing flexible and complex pattern matching services so that you don't have to write a lot of repetitive code to look for one condition after another. For example, you've likely written dozens - or hundreds - of lines of code trying to validate a phone number or an email address. A regular expression can do that in one line.

You can think of regular expressions as something along the lines of strtran(), using a pattern matching expression for the translation of complex expressions.

For example, with strtran() you can look for one string in another, and replace it with a third. Suppose you had a field full of phone numbers, and somebody had mistakenly entered a bunch of the '404' area codes with '464'? It'd be easy to replace them:

```
cPhone = '464-555-1234'
strtran(cPhone, '464-', '404-')
```

returns

```
404-555-1234
```

The problem with strtran is that you can only search for literal strings. What if one person entered '464' and another person entered '484' - both of them meaning to enter '404'?

There isn't any way to do a single search on both, sort of like

```
strtran(cPhone, '4*4-', '404-')
```

where the '*' would mean either a 6 or an 8. You'd have to do two separate searches.

Wouldn't it be nice if you could use a wildcard, sort of like when you used "*" and "?" in DOS:

```
dir *.txt
dir BUDGET19??.xls
```

back in the day?

With a “regex”, you can! In fact, you're not limited to brute force wildcards. You can use a pattern, which acts as a 'smart' wildcard. So here we'd use a pattern,

```
"4[6|8]4-...-...."
```

where the

```
[6|8]
```

means 'either 6 or 8 in the second position, between the first and second 4s', and the

```
.
```

is a 'any character' wildcard. Then, using the regexp() function (I'll discuss where that 'regexp()' function comes from shortly), you could determine if one string was in another:

```
cPattern = '464-555-1234'
```

```
? regexp(434-555-1000, cPattern)
```

returns

```
.F.
```

while

```
? regexp(464-555-1000, cPattern)
```

returns

```
.T.
```

Regular expressions can do more than pattern matching. A full library would also include pattern matching, positional searching (similar to substr()), and replacements (similar to strtran().) The library used in this article simply does pattern matching, not pattern replacement. In order to update the field with the corrected version, you'd need to search for the offending match(es) and then translate the errant values separately. For example, suppose you had potentially bad values in field cPhone1 and wanted to fill cPhone2 with all good values.

```
replace cPhone2 with ;
    iif(regexp(cPhone1,cPattern), ;
        '404'+ substr(cPhone1,4), ;
        cphone1) ;
    all
```

This is a simple, nee trivial, example. Regular expressions can match and validate complete Zip codes, phone numbers, Social Security Numbers, email addresses, URLs, part numbers, and so on. Often one line of code can replace dozens or more. So that's how regular expressions can help you become more productive.

2. How to implement regular expressions in VFP

You likely know that there isn't a 'regexp()' function built into Fox. So how can we implement regular expressions in VFP, and, more specifically, where did it come from in the example above?

Windows Scripting Host

One way to implement regular expressions is through the Windows Scripting Host (WSH). which contains its own regular expressions parsing mechanism.

```
lpara lcPhone
lcPhone = allt(lcPhone)

local loRegExp
loRegExp = createObject("VBScript.RegExp")
loRegExp.Pattern = "4[6|8]4-...-...."

llRetVal = loRegExp.test(lcPhone)
```

I don't care for the WSH myself, for several reasons. First, due to a number of gaping security holes in its early days, I stayed away from it. I still have my doubts about its security, but that's likely as much my own foible as anything else.

Second, and more importantly, using the WSH requires your app to be hooked into Windows; each time you connect to a Windows component, you're at risk for your app breaking when Windows is updated.

The regexp() function is available through a third-party library that doesn't need to be connected to Windows at all; throw it and a few supporting files into your app's folder, and you're all done. Another example of "Fire and Forget" that I love so much about Fox.

RegExp.FLL library

Craig Boyd (many people are not aware of this, but there is not just one person named 'Craig Boyd', it's actually a label for a set of identical triplets, none of whom actually ever sleeps) wrote a regular expressions library for VFP that provides pattern matching services. He described how he did it here:

```
http://www.sweetpotatosoftware.com/SPSBlog/PermaLink,guid,91241006-595a-487d-ac06-d0fc1fc71632.aspx
```

The regexp() function is available from this VFPRegExp.fll library.

The library requires some C++ runtimes; they are included in this article's source code along with the FLL. They are packaged in two zip files, regexp.zip and microsoft.vc80.crt.zip.

To start working with regular expressions yourself, download the zips and place them in a folder, say, 'RegExEx' (for REGular EXpressions EXamples, get it?). Unzip both, creating a folder structure like so:

```
regexex\
    Microsoft.VC80.CRT\
        msvcp80.dll
        msvcr80.dll
        msvcm80.dll
        Microsoft.VC80.CRT.manifest
    regexp.fll
```

Then start VFP, CD into regexex\, and follow along. You can duplicate this folder structure as part of your app; no registration with Windows or other tomfoolery is necessary!

3. Regular expressions pre-built examples

There is no practical limit to the variety of pattern matching you can do with regular expressions.

Additionally, time and space limit us to offering the barest of examples.

So, before I to any further, I want to mention that even though this article will only scratch the surface, you may feel a bit intimidated, thinking that it wouldn't make sense to become an expert at this demanding syntax for an occasional use. And you'd be wrong!

There is an amazing library of patterns at

```
http://regexlib.com/Search.aspx?Aspx
```

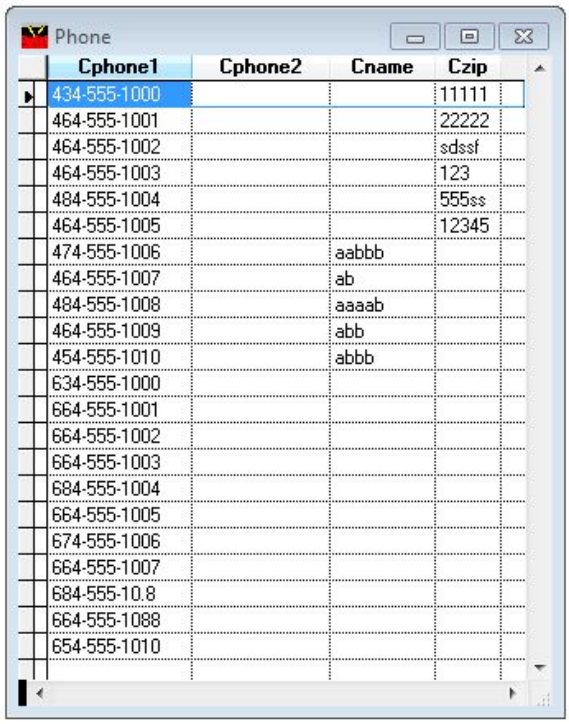
Enter a type of pattern you want to search for in the Keyword text box, tweak the desired results via the other controls, and hit Search. You'll be greeted with results for a large variety of patterns. See **Figure 1**.

Figure 1. The regexlib.com pattern library.

So, my suggestion for you is to scan this article to get the concept and become familiar with the basic syntax. Then, instead of spending hours learning the ins and outs of every technique to engineer the pattern you need, use the library to do the heavy lifting for you.

4. Regular expressions syntax and examples

In the examples below, I'm going to use a simple table that contains a list of phone numbers, a few zip codes, and some random character strings. See **Figure 2** for a birds-eye view.


Phone

Cphone1	Cphone2	Cname	Czip
434-555-1000			11111
464-555-1001			22222
464-555-1002			sdssf
464-555-1003			123
484-555-1004			555ss
464-555-1005			12345
474-555-1006		aabbb	
464-555-1007		ab	
484-555-1008		aaaab	
464-555-1009		abb	
454-555-1010		abbb	
634-555-1000			
664-555-1001			
664-555-1002			
664-555-1003			
684-555-1004			
664-555-1005			
674-555-1006			
664-555-1007			
684-555-10.8			
664-555-1088			
654-555-1010			

Figure 2. The data for these examples.

Single character matching

The period, as we've seen earlier, matches a single character.

```
browse last for regexp(cPhone1, ;
   "464-...-....")
```

returns all records where the area code is '464'.

Match any one of a set of characters

The brackets, also as we've seen, matches any single one of the characters enclosed inside the brackets.

```
browse last for regexp(cPhone1,  ;
   "464-...-...[12345]")
```

returns any 464 number that ends in 1, 2, 3, 4, or 5.

```
464-555-1001
464-555-1002
464-555-1003
464-555-1005
```

(Why isn't '1004' in the list? Because the area code for 1004 is 484.)

Ranges

You can use a range inside the brackets as well:

```
browse last for regexp(cPhone1, ;
   "464-...-...[5-9]")
```

Ranges can include numbers, lower case characters or upper case characters. And, yes,

regular expressions ARE case sensitive! You can ignore case by preceding the pattern with

```
(?i)
```

like so

```
(?i)[a-z]
```

Anything but

The carat, ^, negates a condition, like so:

```
[^3]
```

so

```
browse last for regexp(cPhone1,  ;
   "464-...-...[^12345]")
```

returns

```
464-555-1007
464-555-1009
```

(Again, 1008 doesn't have a 464 area code.)

Either or

The pipe, |, separates two options and matches either one or the other. So

```
browse last for regexp(cPhone1, ;
   "4[6|8]4-...-....")
```

finds all 464 and 484 area codes.

Beginning and end matches

The metacharacters ^ and $ force a match at the very beginning or very end of a string.

```
browse last for regexp(cPhone1,  ;
   "^464-...-....")
```

returns all numbers that begin with '464'. Then

```
browse last for regexp(cPhone1,  ;
   "...-...-..08$")
```

returns all numbers that end in '08'. And

```
browse last for regexp(cPhone1,  ;
   "^[4|6]..-...-..08$")
```

returns all numbers that start with '4' or '6' and end in '08'.

Repetitions

The braces, {}, match a number of instances inside the braces of the string just before the braces. For example,

```
abcz{3}
```

will match the string

```
abczzz
```

but not strings with 2 or 4 z's.

Supposing you had a table with a Name field. To find all values with an 'abb' string:

```
browse last for regexp(alltrim(cName), ;
   "ab{2}")
```

Note the use of 'alltrim()' around the field. Without this, the regexp() function will look for an exact match of the (three character) string 'abb' in a ten character field, and not find any matches. With alltrim(), you're trimming leading and lagging spaces, and thus have a better chance of finding that match.

Add a comma to match any number equal to or greater than the number inside the braces. This:

```
abcz{2,}
```

will match

```
abczz, abczzzz and abczzzzzz
```

Special metacharacters

The \d metacharacter matches any single digit. It's a replacement for [0-9].

```
browse last for regexp(cZip, "\d{5}")
```

The \w metacharacter does the same for alphabetic characters.

There's more!

These examples demonstrate some of the more common regex syntax, but in no way are a comprehensive list. The definitive guide is here:

http://www.regular-expressions.info/reference.html

And the RegEx Pal website,

http://regexpal.com/

provides a handy syntax checker.

Breaking apart a couple of common patterns

Let's put together what we've learned so far into a couple of common examples.

U.S. Social Security Number

The U.S. SSN has the pattern of 999-99-9999. This one should be pretty easy.

The pattern

```
\d{3}
```

is used for the first segment, and similar patterns for the other two segments. Combine them with hyphens, and you get:

```
\d{3}-\d{2}-\d{4}
```

Any string of the form NNN-NN-NNNN will match. Note that this doesn't validate the number to be valid, but it forces the match on the structure.

Credit Card Number

A similar pattern can be used to validate credit card numbers:

```
\d{4}-\d{4}-\d{4}-\d{4}
```

This pattern isn't very intelligent; it just looks for four four-digit strings separated by hyphens. To make it more useful, you could use

```
[345]\d{3}
```

as the first string to ensure that the card number begins with a valid digit. You could further include a

```
6
```

if you wanted to include American Express, but you'd then have to do some extra work at the end because Amex numbers are only 15 digits long.

U.S. Phone Number

To match a US phone number of the form (999) 999-9999, the pattern would look like

```
((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}
```

Not the easiest thing to read, even now. If I had introduced this pattern to you a few pages ago, you would have shaken your head and moved to the next article. But now, we have many of the tools needed to break it apart and analyze it.

The last section, \d{3}-\d{4}, is made up of three pieces:

```
\d{3}
-
\d{4}
```

These three obviously match the 999-9999 part of the phone number. The first part is a bit more complicated. Break it into two pieces and the separator:

```
((\(\d{3}\) ?)
|
(\d{3}-))?
```

where the pipe is the separator. The "\d{3}" generates the '999' part, obviously, and then opening and closing parens come along for the ride. There are two parts so that both "(999)-" and "(999) " work. The "?" metacharacter, heretofore unannounced, matches either zero or one instances of the preceding character (the space or the parents.)

What's next

The work we've done in this article should be enough for you to consider adding regular expression pattern matching to your application the next time you need it. With 15 or 30 minutes of work, you can have simple regexes added to your system, and from there, it's up to you how far you want to take them.

Source code

Source code for this article is contained in three files. The first contains the regex FLL. The second contains the C++ runtime files. And the third contains a small VFP table, phone.dbf, that can be used with the examples in this article.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com

The \s metacharacter matches a single white space character, including space, tab, form feed, and line feed. It is the same thing as specifying [\f\n\r\t\v].

^take\snote$

This expression will match take note but not takenote because the pattern calls for a space between "take" and "note."

\w

The \w metacharacter matches any alphanumeric character, including the underscore. You can use this in place of [A-Za-z0-9_].

^take\wote$

This expression will match takenote, takevote, take8ote, and so on because the pattern states that the fifth position can be any alphanumeric character.

\d

The \d metacharacter matches a digit character. You can use this in place of [0-9].

^\dtakenote$

This expression will match 3takenote, 7takenote, 9takenote and so on.

That wraps up the section on regular expression pattern development. Let's take this new knowledge and apply it to a number of common patterns.

An Analysis of Some Common Patterns

With the fundamentals of pattern building under your belt, think about these more popular general expressions in use today.

US Zip Code (5-digit)

\d{5}

This will match exactly five digits.

US Zip Code (5- or 9-digit)

\d{5}(-\d{4})?

As with the above pattern, the \d{5} will match exactly five digits. The key to this pattern is the (-\d{4})?. Working from the inside out you can see there needs to be four digits preceded by a hyphen. That pattern is then grouped and a ? qualifier is applied to it which says that 0 or 1 matching patterns of four digits will work. With this pattern 27624 and 27624-1234 are both valid. Slick huh?

U.S. Phone Number (999) 999-9999

((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}

This is one of the patterns I used in the opening paragraphs. It looked daunting at the time. It may still look a bit confusing, but at least you should recognize all of the metacharacters I used to construct it. Let's break it down into more manageable parts.

If you start at the end of the pattern you will find \d{3}-\d{4}. This pattern will match exactly three digits followed by a hyphen and then exactly four digits. That part will be responsible for matching the phone number portion 999-9999 of the string. Now let's focus on the ((\(\d{3}\) ?)|(\d{3}-))? subpattern. My eye is drawn to the ? on the end of the subpattern. It will match 0 or 1 instances of the pattern grouped by the parenthesis. This means that a phone number will be a match valid with or without an area code.

On the right side of the | (or) metacharacter is the (\d{3}-) pattern. It will match exactly three digits followed by a hyphen. The right side of the | is the (\(\d{3}\) ?) pattern. It uses the \ metacharacter to specify that a left parenthesis \(and a right parenthesis \) are part of the pattern and should not be considered grouping metacharacters. Between the \(and the \) is \d{3} which will match exactly three digits.

So, (555) 123-4567, 555-123-4567, and 123-4567 all match and will be considered valid U.S. phone numbers. See, that wasn't so bad after all.

U.S. Social Security Number

\d{3}-\d{2}-\d{4}

After that U.S. phone number this one should be easy. It specifies a pattern of exactly three digits followed by a hyphen then exactly two digits followed by a hyphen and exactly four digits. Strings that match would include 123-45-6789, 000-00-0000, and 555-55-5555. Notice, these may not be valid U.S. Social Security numbers but they do match the pattern.

Date

^\d{1,2}\/\d{1,2}\/\d{4}$

This pattern will match a date in the 99/99/9999 format. Starting from left to right, the ^\d{1,2} subpattern specifies that a number at least one digit in length but not longer than two digits must be at the beginning of the string. Next comes a \/ which makes the / act as a literal character. Next comes \d{1,2} again, followed by another \/. Lastly this pattern specifies that exactly four digits must be at the end of the pattern.

Offensive Words

(\bBadWord1\b)|(\bBadWord2\b)|...|(\bbadWordn\b)

This pattern will match any words you specify as offensive and this pattern makes it easy to keep unwanted words from making their way into your data. The \b metacharacter matches any word boundary, such as a space.

(\bdratsl\b)|(\bshoot\b)|(\bdarn\b)

This pattern will match any text stream that contains the word drats or shoot or darn.

Table 2 contains a few more commonly used regular expression patterns.

Basic Credit Card

^(\d{4}[-]){3}\d{4}|\d{16}$

This pattern will match a credit card number in the format of 9999-9999-9999-9999, 9999 9999 9999 9999, or 9999999999999999. Let's break this pattern down from right to left. One the right side of the | (or) we see \d{16} which specifies sixteen digits. That's pretty straightforward. The left side of the | (or) looks a bit more complicated. I'll start with the ^(\d{4}[-]). This specifies a grouped string of four digits followed by a hyphen or a space. Next is {3} which specifies that there must be exactly three 4-digit grouped strings. Next is the \d{4} which specifies the final four digits.

You may have noticed that this pattern doesn't validate the number at all or categorize it by type of card. 4999-9999-9999-9999 is just as valid as 1999-9999-9999-9999 even though no credit card starts with the number 1.

Advanced Credit Card

^((4\d{3})|(5[1-5]\d{2})|(6011))-?\d{4}-?\d{4}-?\d{4}|3[4,7]\d{13}$

This is another credit card pattern but this time we're specifying that the card number must start with a 4, 5, 6, or 7. This pattern matches all the major credit cards including Visa which has a length of 16 and a prefix of 4, MasterCard which has a length of 16 and a prefix of 51-55, Discover which has a length of 16, and a prefix of 6011, and finally American Express which has a length of 15 and a prefix of 34 or 37. All of the 16 digit formats (Visa, MasterCard, and Discover) accept an optional hyphen between each group of four digits.

Let's start with the ^((4\d{3})|(5[1-5]\d{2})|(6011)). It's not as bad as it looks. The first thing to notice is that it is one big group with two OR conditions inside. This group is going to be the definition for the first four digits of the card. The string must start with a group comprised of a "4" followed by exactly three digits (4\d{3}) OR a group comprised of a "5" followed by a 1, 2, 3, 4, or 5, followed by exactly four digits (5[1-5]\d{2}), OR a group comprised of a 6011 (6011).

" You can find out more about the RegEx Class in the Visual Studio .NET help. "

Next is -?, which means that there can be 0 or 1 hyphens following the initial set of four digits.

Next is \d{4}-?, which refers to the second group of four digits. It means that exactly four digits followed by 0 or 1 hyphens are acceptable.

Next is another \d{4}-?, which refers to the third group of four digits. It, too, means that exactly four digits followed by 0 or 1 hyphens are acceptable.

Next is \d{4}, which refers to the fourth group of four digits. It means that exactly four digits are acceptable.

Next is the | (or) which signals the end of the 16 digit pattern and the beginning of the American Express pattern. This pattern, 3[4,7]\d{13}$ means that the string must start with "34" or "37" followed by 13 digits. In this pattern, spaces and hyphens are not acceptable in American Express card numbers.

Using Regular Expressions in Visual Studio .NET

Now that you've seen the fundamentals of building a regular expression pattern, and you've reviewed a number of popular patterns, it's time to find out how to implement a regular expression in Visual Studio .NET.

Visual Studio .NET supports regular expressions in ASP.NET via the RegularExpressionValidator control and in code via the RegEx class.

ASP.NET

While there is more than one way to use regular expressions in ASP.NET, the most common is to use the RegularExpressionValidator control.

A developer uses the RegularExpressionValidator control to make sure the entry in a watched control conforms to a specified regular expression. This allows you to check an entry against a pattern such as a U.S. Social Security number, telephone number, e-mail address, etc.

The following snippet lists the HTML for a RegularExpressionValidator control watching over a textbox containing U.S. phone number information.

```
<asp:RegularExpressionValidator
  id="regPhoneNumber"
  runat="server"
  ErrorMessage=
      "Please enter a valid phone number!"
  ControlToValidate="txtPhone"
  ValidationExpression=
      "((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}">
</asp:RegularExpressionValidator></P>
```

The Property window in Figure 1 shows the regPhoneNumber Control.

Click for a larger version of this image.

Figure 1: The properties for the regPhoneNumber RegularExpressionValidator control.

Recognize that pattern? That's the same U.S. Phone pattern you saw earlier. Fortunately for me I didn't have to type that into the ValidationExpression property. I used the Regular Expression Editor (see Figure 2), which has a number of pre-built Regular Expressions available for you to select from including e-mail addresses, Internet URLs, U.S. Social Security number, U.S. zip code, as well as international phone and postal code formats. Of course, you can also develop you own custom expressions.

Click for a larger version of this image.

Figure 2: The Expression Editor provides a number of expression templates for you to use.

The System.Text.RegularExpressions Namespace

In addition to the ASP.NET RegularExpressionValidator control, you can also take advantage of the classes contained in the .NET Framework regular expression engine. These classes are contained in the System.Text.RegularExpressions namespace.

The RegEx Class

The RegEx class handles the majority of the work in the System.Text.RegularExpressions namespace. The constructor of this class is critical because it contains the most important element of a regular expression, the pattern. You can code the constructor in one of three ways.

```
'Passing no parameters
RegEx()

'Passing the string pattern
RegEx(pattern)

'Passing the string pattern and option settings
RegEx(pattern,options)
```

The pattern parameter, if passed, needs to be a string. The options parameter, if passed, needs to be a member of the RegExOptions enumeration.

The RegExOptions enumeration contains the options that you can set when you create a RegEx object. IgnoreCase is a commonly used option that overrides RegEx's default case-sensitivity behavior. Include this option if you want to have a case insensitive regular expression. Another way to specify case insensitivity is to add (?i) to the beginning of the pattern.

```
^(?i)[a-z]{3}$
```

This expression will match abc, AbC, and ABC.

Since matching is one of the most commonly performed operations, let's start with it. The code below determines if you've entered a valid U.S. phone number into a textbox.

```
Dim oRegEx As Regex = New _
Regex("((\(\d{3}\) ?)|
(\d{3}-))?\d{3}-\d{4}")
Dim x As Boolean
x = oRegEx.IsMatch(Me.TextBox1.Text)

If x Then
   MessageBox.Show("Valid!")
Else
   MessageBox.Show("Invalid!!!")
End If
```

The IsMatch()method returns true if it finds the pattern in the passed string, false otherwise. The static version of IsMatch accepts three parameters: the string passed in, the pattern to check it against, and the RegExOptions required.

```
If Regex.IsMatch(Me.TextBox2.Text, _
 "[a-z]{3}", RegexOptions.IgnoreCase)
Then
```

Match Object

The RegEx.Match method returns a Match object that provides detailed information about a match, including whether or not a match was found, the value (the text matched), the index (position in the searched string), and length of the string matching the pattern.

```
Dim SearchString As String _
  = "A cobra is a venomous snake!"
Dim Pattern As String = "\bve\w*"
Dim oMatch As Match

oMatch = Regex.Match(SearchString, _
  Pattern, RegexOptions.IgnoreCase)

If oMatch.Success Then
  MessageBox.Show(oMatch.Value)
  MessageBox.Show(oMatch.Index)
  MessageBox.Show(oMatch.Length)
End If
```

The pattern in the above code will match any word (note the \b metacharacter) that begins with "ve". It matches on the word venomous, therefore

oMatch.Success return true. oMatch.Value contains "venomous," oMatch.Index contains 13, and oMatch.Length contains 8.

Either RegEx.IsMatch() or Match.Success will work if you're only looking for a single match. What if you want to find all of the occurrences in a string that match the pattern? This is where NextMatch() comes in.

NextMatch() will return the next match in the searched string starting from the end of the current match. You can place NextMatch() inside a loop to iterate through a searched string to find all the occurrences of the pattern.

```
Dim SearchString As String = _
  "A cobra is a very, very venomous snake!"
Dim Pattern As String = "\bve\w*"
Dim oMatch As Match
Dim MatchHits As Integer = 0

oMatch = Regex.Match(SearchString, _
  Pattern, RegexOptions.IgnoreCase)

Do While oMatch.Success
  MatchHits = MatchHits + 1
  oMatch = oMatch.NextMatch()
  If oMatch.Success Then
    MessageBox.Show(oMatch.Value)
    MessageBox.Show(oMatch.Index)
    MessageBox.Show(oMatch.Length)
  End If
Loop
```

The previous code will find three matches to the pattern, starting with the first occurrence of the word "very." A loop then begins based on the success of the first match. The NextMatch() method is called within the loop to find the next pattern match.

While this technique may work for you in some circumstances, it has limitations. The technique doesn't provide a way to index any match or provide any metadata about the match set, such as a match count. In a previous example the code maintained a MatchHits variable manually.

MatchCollection

If you need the ability to iterate through the match set, if you need to know how many matches occurred, or if you need to be able to do ad-hoc indexing into the match set, then MatchCollection is the object for you.

```
Dim SearchString As String = _
  "A cobra is a very, very venomous snake!"
Dim oMatch As Match
Dim oMatchCollection As MatchCollection
Dim oRegEx As New Regex("\bve\w*")

oMatchCollection =
oRegEx.Matches(SearchString)

MessageBox.Show(oMatchCollection.Count)
For Each oMatch In oMatchCollection
  MessageBox.Show(oMatch.Value)
  MessageBox.Show(oMatch.Index)
  MessageBox.Show(oMatch.Length)
Next
```

The above code does almost exactly the same thing as the preceding code sample that used NextMatch(). The difference is that this code uses the Matches method to create a MatchCollection object.

Replacement Strings

Replacement does exactly what you expect it to do?find a piece of text that matches a pattern and replace it with another.

```
Dim SearchString As String = _
  "A cobra is a very, very venomous snake!"
Dim Pattern As String = "\bven\w*"
Dim ReplacementString As String = "friendly"
Dim NewString As String

NewString = _
 Regex.Replace(SearchString, Pattern, _
ReplacementString)
```

The above code will find all occurrences of the word "venomous" and replace it with the word "friendly."

The replacement capabilities of regular expressions are limitless. You could read an HTML file and remove all of the bold tags (<B> and </B>) with a simple Regex.Replace() call.

```
Dim SearchString As String = _
  "A cobra is a very, <B>very</B> venomous
snake!"
Dim Pattern As String = "(<b>)|(</b>)"
Dim ReplacementString As String = ""
Dim NewString As String

NewString = _
Regex.Replace(SearchString, Pattern, _
ReplacementString,
RegexOptions.IgnoreCase)
```

This code results in NewString containing "A cobra is a very, very friendly snake!"

```
(<b>)|(</b>)|(<i>)|(</i>)
```

This pattern expands on the previous example to remove italics tags as well.

Summary

My goal at the beginning of this article was to introduce you to the world of regular expressions and how to use them in Visual Studio .NET. While they can look complex and overwhelming to the uninformed, once you break them apart they're really not that bad to work with and they provide a powerful tool to add to your development toolbox.

Resources

You will find plenty of help and resources on regular expressions on the Web. Here are just a few of the resources you'll find:

Software

RegExDesigner (freeware)

http://www.sellsbrothers.com/tools/

RegexDesigner.NET is a powerful visual tool for helping you construct and test .NET Regular Expressions. When you are happy with your regular expression, RegexDesigner.NET lets you integrate it into your application through native C# or VB .NET code generation and compiled assemblies (usable from any .NET language).

Web Site

www.regexlib.com

A Web site with an extensive collection of contributed regular expression patterns. You can submit your patterns and/or test your patterns before you implement them with the Regular Expression Tester.

Books

Mastering Regular Expressions, Second Edition

Jeffrey Friedl

Regular Expressions with .NET (ebook)

by Dan Appleman

Visual Basic .NET Text Manipulation Handbook: String Handling and Regular Expressions

By Paul Wilton, Craig McQueen, François Liger

There are three categories of differences between SQLite and VFP SQL. These are (1) language differences, (2) engine differences, and (3) implementation differences.

Language differences

The whole point of SQL is to have a standardized query language that works across platforms, hardware, languages. Thus, it may come as a surprise to some that "All SQLs are equal, but some are more equal than others."

There are several standard SQL commands not implemented in SQLite. Furthermore, the SQLite function set is much smaller than VFP's.

Commands - Joins

Visual FoxPro supports full outer joins. Suppose you've got a parent table and a child table, such as Person and Pet, where a Person could have one or more Pets, but a Pet may or may not belong to a specific Person. In VFP, you can write a query to find all Persons with one or more Pets, like so:

```
select * from person join pet on ;
pet.iidperson = person.iidperson
```

This pulls only records from Person and Pet where a Person has a Pet. It ignores the Persons who are petless as well as the Pets who are strays.

You can also write a query to find all Persons, whether they have Pets or not, but ignoring strays:

```
select * from person left join pet on ;
pet.iidperson = person.iidperson
```

You can write a query to find all Pets, regardless if they have an owner or if they're a stray, but ignoring Persons who don't have a Pet:

```
select * from person right join pet on ;
pet.iidperson = person.iidperson
```

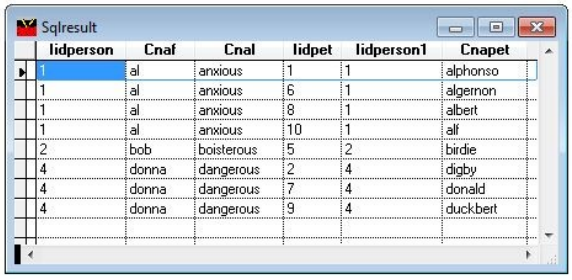
And, finally, you can write a query to find all Persons and all Pets, regardless if they have a matching record in the other table, like so:

```
select * from person full join pet on ;
pet.iidperson = person.iidperson
```

You execute SELECTS like this with SQLite via SQLEXEC(). (See the March article for details and multiple examples.) For example,

```
m.lcXN = "DRIVER={SQLite3 ODBC Driver};" ;
   + "Database=f:\PersonPet;"
liH = sqlstringconnect(lcXN)
=sqlexec(lih, "select * from person ;
   join pet ;
   on pet.iidperson = person.iidperson")
```

will produce a cursor named "sqlresult" that can be browsed. This cursor contains all Persons who have one or more Pets, but no petless Persons, nor any strays. See **Figure 1**.


Sqlresult

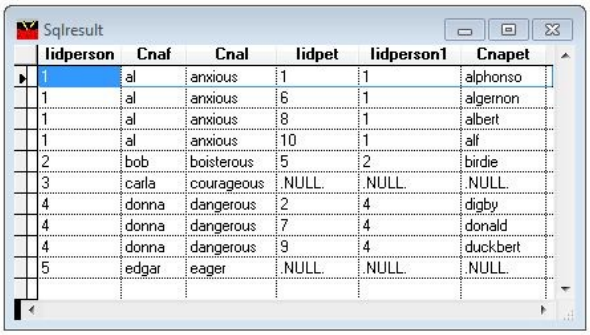
iidperson	Cnaf	Cnal	iidpet	iidperson1	Cnapet
1	al	anxious	1	1	alphonso
1	al	anxious	6	1	algernon
1	al	anxious	8	1	albert
1	al	anxious	10	1	alf
2	bob	boisterous	5	2	birdie
4	donna	dangerous	2	4	digby
4	donna	dangerous	7	4	donald
4	donna	dangerous	9	4	duckbert

Figure 1. A join executed in SQLite.

Here's the first big difference between VFP and SQLite: SQLite only supports left joins, not right or full joins. So you can find out matches, where there is a Person and a Pet, using a regular join. You can also find Persons with or without Pets, like so:

```
select * from person left join pet on ;
pet.iidperson = person.iidperson
```

and shown in **Figure 2**.


Sqlresult

iidperson	Cnaf	Cnal	iidpet	iidperson1	Cnapet
1	al	anxious	1	1	alphonso
1	al	anxious	6	1	algernon
1	al	anxious	8	1	albert
1	al	anxious	10	1	alf
2	bob	boisterous	5	2	birdie
3	carla	courageous	.NULL.	.NULL.	.NULL.
4	donna	dangerous	2	4	digby
4	donna	dangerous	7	4	donald
4	donna	dangerous	9	4	duckbert
5	edgar	eager	.NULL.	.NULL.	.NULL.

Figure 2. A left join executed in SQLite.

In order to find Pets with or without owners, you can't use a right join:

```
select * from person right join pet on ;
pet.iidperson = person.iidperson
```

If you try to execute this, you'll get no response. Specifically, no 'sqlresult' cursor will be created, nor will you see an error message. Instead, if you need a right join, you'll need to reverse the join syntax, like so:

```
select * from pet left join person on ;
person.iidperson = pet.iidperson
```

Not a big deal, but one that can bite you if you're not aware, and try to use the 'right' keyword.

Full joins aren't recognized either, but you can get the same result through a bit of trickery. Since you're looking for all possible combinations, you'll need to do a pair of left joins, reversing the syntax as shown earlier for the second, and then do a UNION to merge the results:

```
=sqlexec(lih, "select cnaf, cnal, cnapet ;
   from person left join pet ;
   on pet.iidperson = person.iidperson ;
union ;
select cnaf, cnal, cnapet ;
   from pet left join person ;
   on person.iidperson = pet.iidperson")
```

The first left join finds all Persons with or without pets; the second finds all Pets with or without owners. The UNION clause merges the two interim result sets and removes the duplicates. Unfortunately, as shown in **Figure 3**, the UNION trickery causes each field to be converted to a memo field in the cursor, so you may need to do more work, depending on your ultimate goal.

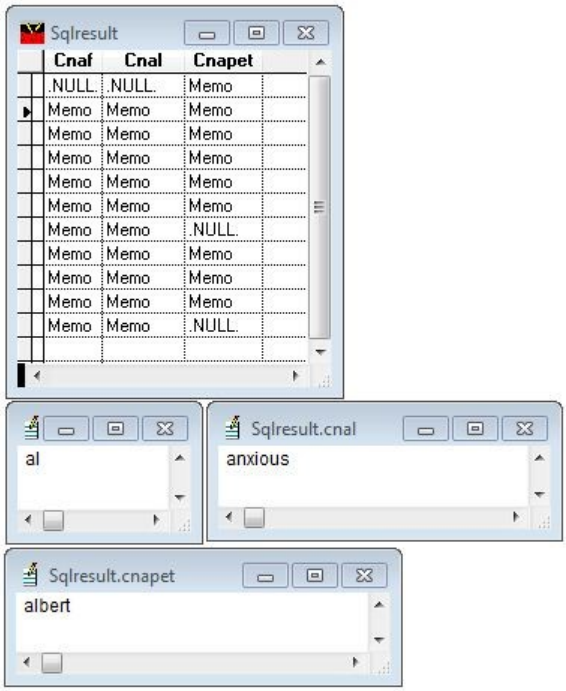


Figure 3. Fooling SQLite into doing a full join.

Commands – Alter Table

The other big difference is that VFP's Alter Table command is much more robust.

In VFP, you can:

- add a new column,
- change an existing column,
- delete (drop) a column,
- rename a column,
- add a candidate index,
- delete (drop) a candidate index,
- add a primary key,
- delete (drop) a primary key,
- add a foreign key,
- delete (drop) a foreign key,
- set a check (validation rule), and
- set an error message if a check fails.

Additionally, columns can:

- be set to be auto-incremented,
- have a default value,
- be prevented from being translated to a different code page.

By contrast, SQLite's Alter Table functionality rather limited. You can:

- rename a table
- add a column

Let's take a look. Suppose we realized we needed another table in our Person/Pet database for all of the paraphernalia associated with taking care of a critter. We add a table called 'gear' to the database. Not sure what fields we need yet, we just define a single primary key:

```
=sqlexec(lih, "create table gear (iidgear
bigint)")
```

And prove that it worked:

```
=sqlexec(lih, "insert into gear (iidgear)
values (1)")
```

Of course, as soon as we hit the Enter key, the realization that we needed a descriptor hit, so we used Alter Table to add the new column:

```
=sqlexec(lih, "alter table gear add column
cdesc char(20)")
```

And it works just fine:

```
=sqlexec(lih, "insert into gear (iidgear,
cdesc) values (2,'leash')")
```

There was already a row in the table without a value in the cdesc field, so let's take care of that as well:

```
=sqlexec(lih, "update gear set cdesc =
'sweater' where iidgear = 1")
```

Time passes, and we decide that we preferred a different name for the 'gear' table. So we'll rename it to 'equipment', like so:

```
=sqlexec(lih, "alter table gear rename to
equipment")
```

Unfortunately, THEN we discover that we can't rename columns in a SQLite table, so the primary key, iidgear, is now a misnomer, so we rename the table back:

```
=sqlexec(lih, "alter table equipment rename to
gear")
```

And all is good.

There are other differences between SQLite's implementation of SQL and the full language, but these two (Join, Alter Table) that most VFP developers need to be concerned with.

Functions

SQLite has 32 functions. Visual FoxPro has more than 32 that start with the letter 'A'. (Seriously! Check it out!)

As a result, there is a great disconcordance between SQLite and VFP's function set. A few in SQLite don't map to any in VFP, and a great many in VFP don't have correlations in SQLite.

More importantly, there are a few that do match up, albeit sometimes inexactly. It's handy to have a chart of how those do match up, and what gotchas exist. See **Table 1**.

Table 1. Mapping of VFP and SQLite functions.

VFP	SQLite	Comments
abs	abs	Returns absolute value of argument.
	coalesce ifnull	Returns first non-NULL argument.
len	length	Returns lengths of argument.
lower	lower	Returns lower case version of string argument.
ltrim	ltrim	Returns string after removing spaces from left side of string. (SQLite version has additional functionality.)
max	max	Returns largest argument. Serves as an aggregator with only a single argument.
min	min	Returns smallest argument. Serves as an aggregator with only a single argument.
	nullif	Returns first argument if the arguments are different and NULL if the arguments are the same.
rand	random	Returns a pseudo random integer. Range differs between VFP and SQLite.
strtran	replace	Returns a string formed by substituting string Z for every occurrence of string Y in string X.
round	round	Returns the first argument rounded to the # of digits defined in second argument.
rtrim	rtrim	Returns string after removing spaces from right side of string. (SQLite version has additional functionality.)
soundex	soundex	Returns the Soundex encoding of the argument. Only available in SQLite if SQLITE_SOUNDEX compile time option is turned on.
substr	substr	Returns a substring of input string X that begins with the Y-th character and which is Z characters long.
trim	trim	Returns string after removing spaces from both left and right sides of string. (SQLite version has additional functionality.)
vartype	typeof	Returns data type of argument. VFP: “C”, “N”, “D”, etc. SQLite: "null", "integer", "real", "text", or "blob".
upper	upper	Returns upper case version of string argument.

In order to practice with these functions, you can send dummy SELECT statements to SQLite, using the function as the only argument, like so:

```
=sqlexec(lih, "select abs(-4)")
```

This will return a cursor with a single row, as shown in **Figure 4**.

Figure 4. Practicing with SQLite functions.

With that technique in hand, let's look at the functions listed in Table 1 in more detail.

abs(X)
Works the same in both VFP and SQLite.

coalesce(X, Y, Z...), ifnull(X, Y)
Returns the first non-NULL argument in the argument list, or NULL if all arguments are NULL. Ifnull() is the same as coalesce() but with exactly two arguments.

len/length(X)
Difference function names for the same functionality – returning the length of the string passed as an argument. The following query:

```
=sqlexec(lih, "select *, length(cdesc) ;
   from gear")
```

produces the result set shown in **Figure 5**.

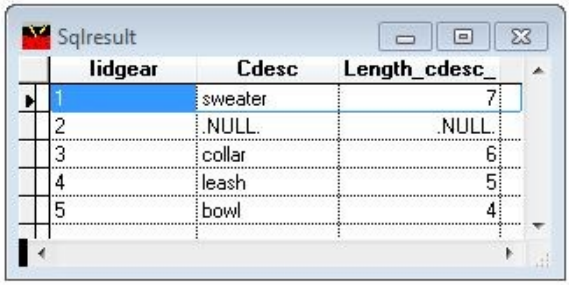


Figure 5. Results of the length() function.

lower(X)
Works the same in VFP and SQLite.

ltrim(X)
Works the same in VFP and SQLite.

max(X,Y), max(X)
Works the same in VFP and SQLite. When passed multiple arguments, returns the largest:

```
=sqlexec(lih, "select max(1,2,3,7,200,6,-4)")
```

The result is shown in **Figure 6**.

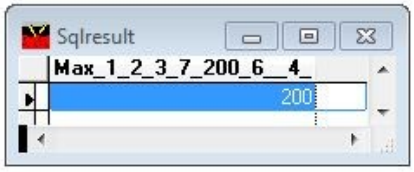


Figure 6. Passing multiple arguments to max().

However, when passed a single argument that is a numeric column in a table, the function returns the value in the row with the largest value:

```
=sqlexec(lih, "select max(iidgear) from gear")
```

The result is shown in **Figure 7**.

Figure 7. Passing a single argument to max().

min(X, Y), min(X)
The min() function works identically to max() except that it returns the lowest value.

nullif(X, Y)
In the same ballpark as coalesce() and ifnull(), this one returns the first argument if the arguments are different and NULL if the arguments are the same.

rand/random()
Works nearly the same in VFP and SQLite. The range of values that the random number is pulled from varies.

In VFP, the random number is a decimal between 0 and 1. It can contain up to 14 decimal places, so when multiplied by a power of ten in order to produce an integer, random numbers available range from 1 to approximately 400 quadrillion.

In SQLite, the random number ranges from approximately -9223 quadrillion to +9223 quadrillion.

Additionally, the VFP rand() function can be seeded with an argument in order to change the start of the random number sequence generated.

rtrim(X)
Works the same in VFP and SQLite.

strtran/replace(X,Y,Z)
Returns a string formed by substituting string Z for every occurrence of string Y in string X. For example,

```
=sqlexec(lih, "select *,
  replace(cdesc,'a','A') as cdescA from gear")
```

will return 'collAr' from a field that contains 'collar'. This is the same functionality as 'strtran()' in VFP.

round(X,Y)
Works the same in VFP and SQLite.

soundex(X)

Soundex is an algorithm for encoding strings as they are pronounced in English, so that words that are spelled differently but sound the same ('homophones') can be compared and sorted.

Soundex is a function contained in all major SQL databases, including Oracle, Microsoft SQL Server, MySQL, and, of course, SQLite and Visual FoxPro. As such, the implementation is the same in both VFP and SQLite. **Figure 8** shows the results of selecting the soundex value of the cDesc field from the gear table in both SQLite (top) and VFP (bottom.)

```
=sqlexec(lih, "select *,
  soundex(cdesc) from gear")
select *, soundex(cdesc) from gear
```

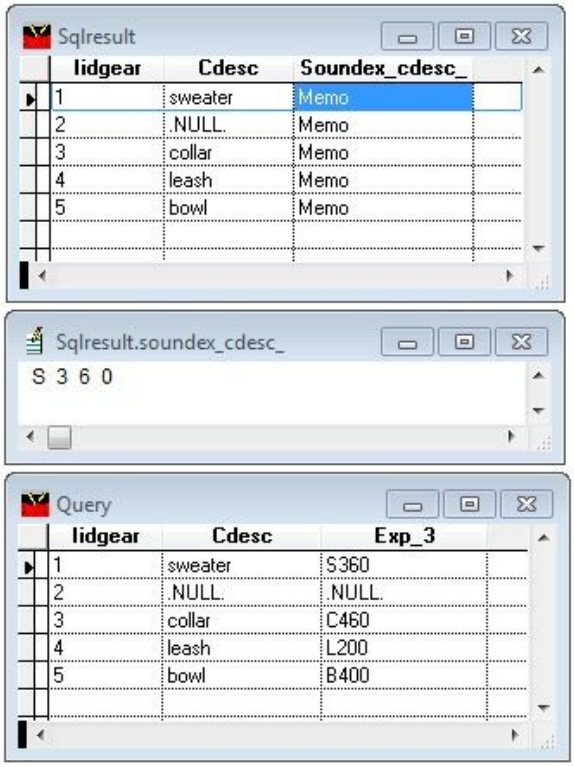


Figure 8. Soundex() results in SQLite and VFP.

Note that Soundex() may not be part of the native SQLite implmentation – it is only available if the SQLite complile-time option, SQLITE_SOUNDEX, is set when the SQLite exectutable is built. Soundex() is part of the SQLite ODBC driver.

substr(X,Y,Z)

Works the same in VFP and SQLite. Returns a string selected from the string X that begins with the Yth character and that is Z characters long. Thus,

```
substr('Visual Fox Pro', 8, 3)
```

returns 'Fox'.

trim(X)

Works the same in VFP and SQLite.

vartype/typeof(X)

Returns the type of data passed in the argument. VFP returns a single upper case character that maps to the data type (for example, "C" for Character or Memo, D for Date.) See the VFP Help file for the complete list. SQLite returns one of the following text strings: "null", "integer", "real", "text" or "blob". The SQLEXEC() command produces the top half of **Figure 9** while the SELECT command produces the bottom half of **Figure 9**.

```
=sqlexec(lih, "select *, ;
   typeof(cdesc) from gear")
select *, vartype('cdesc') from gear
```

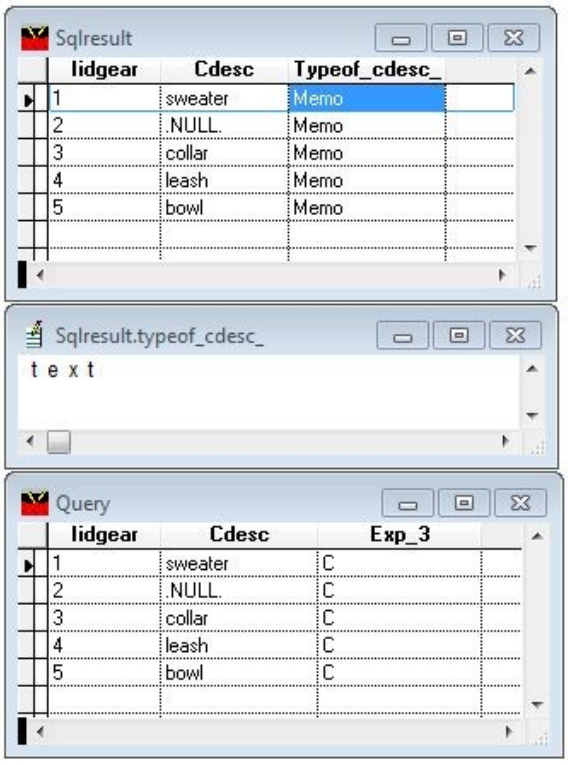


Figure 9. Comparing SQLite's typeof() and VFP's vartype() functions.

upper (X)

Works the same in VFP and SQLite.

Engine differences

Some SQLite functions don't have any direct matches with VFP. See Table 2 for a listing, plus a brief description of what they do.

Table 2. SQLite functions with no corresponding function in VFP..

SQL Function	Description
changes	Returns the number of rows in the database that were

	affected by the last successful INSERT, DELETE or UPDATE statement.
glob	Returns the number of rows where Y is “like” X.
hex	Returns the upper-case hexidecimal translation of the argument.
last_insert_rowid	Returns the row ID of the last row inserted into the database.
like	Returns patterns of the type Y matching the argument X.
load_extension	Strictly not a function, as it doesn't return a useful value. Simply loads a SQLite extension.
quote	Returns a string that is translated so that it can be included in a SQL statement.
randomblob	Returns a string of N random bytes.
splite_compileoption_get splite_compileoption_used	Wrappers around the get and used functions.
sqplite_source_id	Returns a string that contains the version of the source code used to build the SQLite library.
sqlite_version	Returns a string that contains the version of the SQLite library that is running.
total_changes	Returns the number of changes made by the INSERT, DELETE and UPDATE statements since the database connection was opened.
zeroblob	Returns an empty BLOB N bytes long; used to reserve space for a BLOB.

Engine differences

The key engine difference between VFP and SQLite is that SQLite data types are dynamic. In VFP, a column is defined as a specific data type (such as character or date), and from then on, only that type of data can be put into that column.

In SQLite, the data type can change within a column; row 1 can contain a date value while the same column in row 2 can contain a numeric (“integer”) or a character (“text”) value.

Data types are organized into “storage classes”. For example, the integer storage class has six data types – depending on the size of the integer (one byte long, two bytes long, all the way up to eight bytes.) The differences between various data types are for all practical purposes transparent to the user.

Implementation differences

The key implementation difference between VFP and SQLite is that the latter is not multi-user. Multiple users can read from the database, but only one can write. This is because when SQLite writes to the database, it locks the entire database.

As a result, unlike other SQL databases that are inherently multi-user (via locks placed on single records), only a single user can access the database at a single time.

Thus, SQLite is best used for three groups of applications:

- with only one user. This user may be an interactive user in the case of a desktop application, or the Web user in the case of a Web application.

- with multiple read-only users but only one user who will write to the database.

- with more than one regular user who is writing to the database, but where the lock of the database during writes doesn't pose a problem, because a second attempted write can be detected and handled.

Source code:

Source code for this article is contained in two files. The first contains the VFP tables for Person, Pet and Gear. The second contains the SQLite database, PersonPet, which contains all three tables.

Author Profile

Whil Hentzen is an independent software developer based in Milwaukee, Wisconsin (as opposed to Milwaukee, Minnesota, as many people think.) His writing has killed many trees over the years, but none since 2007. He has realized he really sort of misses it. You can reach him at whil@whilhentzen.com