January, 1998

## Advisor Answers

Visual FoxPro 3.0 and 5.0

Q: I would like to create a combo box that functions exactly like the FoxPro help index, that is, when the user types in a value, that value is automatically displayed in the list. If the user just types in the first letter of the entry, that entry is displayed and when the next letter is typed in, the entry that matches those two letters is selected. I have tried creating a text box and list box where the ControlSource of the list is the value of the textbox. The value in the list box is selected, but only after the user has typed in the exact match of the entry. I would like the value to be highlighted when, for example, the user types in the first three letters of the entry.  Instead it is only highlighted when the whole word is typed in.  What am I missing to make this work?

–Judith Barer (via CompuServe)

A: The kind of behavior you want is called "incremental search." VFP's combos and lists have it natively - typing in a listbox always moves you to the first item that matches what you've typed. (If you don't see this behavior, increase the value of the system variable _DBLCLICK, which controls the amount of time you can wait between keystrokes.) However, the native behavior doesn't show you exactly the characters typed so far, the way Help's Index tab does. You need to combine a textbox with another control to get both the echo and the highlight.

This is one of those cases where VFP's ability to subclass its controls and add code to them makes what seems like a difficult task pretty easy. This is a combination you're likely to want to use repeatedly once it works, so start by creating a subclass of VFP's container class.

Add a textbox (txtSearchString) and a listbox (lstData) to the container, giving them equal width and lining them up vertically. Set the BorderWidth of the container to 0, so the two controls look independent. Figure 1 shows the new container class in the Class Designer.

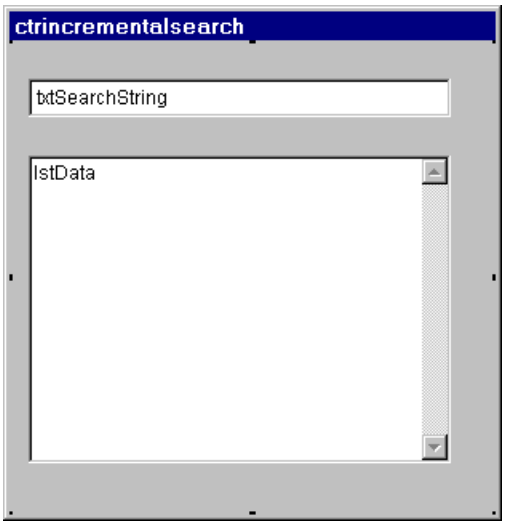


Figure 1. Incremental Search Container Class - The class contains a textbox and a listbox, which work together to provide the kind of incremental search capability offered by Help's Index tab.

By default, string matching in VFP is case-sensitive. Although you'll generally want this control to be case-insensitive, I could imagine some cases where you'd want the default of case-sensitivity. So add a custom property, lCaseSensitive, to the container, which determines whether or not it should be case-sensitive. It defaults to .F., meaning that searches should consider "A"="a".

Set the Value of the textbox to the empty string, so we start with nothing in the textbox.

There are two things a user can do when using this control. He can type into the textbox or he can navigate in the list (using keyboard or mouse). Each calls for a response.

When the user types into the textbox (or otherwise edits the data there), we need to search the list to find the first matching item. Add a custom method called Search to the container class and set the textbox's InteractiveChange method to call the Search method:

```
THIS.Parent.Search()
```

The Search method does the hard work, but even that isn't too complicated. Here's the code:

```
* Search the list for the first entry that matches the
* user's entry in the textbox.

LOCAL cSearchString, cOldExact

cSearchString = TRIM(THIS.txtSearchString.Value)
IF NOT THIS.lCaseSensitive
   cSearchString = UPPER(cSearchString)
ENDIF

cOldExact = SET("EXACT")
SET EXACT OFF

WITH THIS.lstData
   LOCAL cItem, nIndex
   FOR nIndex = 1 TO .ListCount-1
```

```
      cItem = .List[nIndex]
      IF NOT THIS.lCaseSensitive
         cItem = UPPER(cItem)
      ENDIF
      IF cItem = cSearchString
         EXIT
      ELSE
         cItem = .List[nIndex+1]
         IF NOT THIS.lCaseSensitive
            cItem = UPPER(cItem)
         ENDIF
         IF cItem>cSearchString
            * As long as items are in order,
            * as soon as we pass the search string,
            * we can stop
            EXIT
         ENDIF
      ENDIF
   ENDFOR

   .ListIndex = nIndex
ENDWITH

SET EXACT &cOldExact
RETURN
```

There are several things to note in the code. First, the code assumes that the items in the list are sorted. There's no code that forces this to be the case - when you drop the control onto a form, you need to specify the list's RowSource and ensure that it's sorted.

Next, unfortunately, VFP's ASCAN() function doesn't work on the built-in property arrays like List, so we're forced to search sequentially. (Actually, we could implement a more complex search algorithm, but I'd only do that if the search speed became an issue.)

The good news is that using sequential search makes handling case-sensitivity easier. If lCaseSensitive is .F., the search string is converted to upper case at the beginning of the code. The items in the list are converted as they're accessed.

The most complicated part of this code is in handling search strings that don't match anything in the list. In testing the Help Index, I found that when there's no entry that matches what's been typed, the list lands on the closest item before the search string. That is, if your search string contains "Elton" and your list has consecutive entries "Ellen" and "Ethel", "Ellen" gets highlighted.

So, if the search string doesn't match the current item, we need to check whether the next item is too far. That's what's going on in the ELSE clause.

When we leave the loop (whether by finding a match or not), we set the highlight to the current item. Either we found a match, we found that there was no match and stopped before going too far, or we ran out of items to check, in which case we know that even the last item on the list comes before the search string.

The other thing a user can do is move the highlight in the list, either with the mouse or the keyboard. In that case, we want the textbox to reflect the current list value. Add

another method to the container called Position and set the list's InteractiveChange method to call it:

```
THIS.Parent.Position()
```

The code for Position is extremely simple:

```
* Update the textbox when the user makes
* a change in the list
THIS.txtSearchString.Value = ;
   THIS.lstData.List[ THIS.lstData.ListIndex ]
```

In fact, that code is so simple, you may be tempted to ask why we should even bother to create a custom method for it. Since Search is called from only one place, you might ask the same question about it, too. In both cases, it's because updating one control when the other changes is the responsibility of the container, not the control that changed. In general, when designing classes, it's important to ask who has responsibility for a particular action. One of the goals in a container class like this one is for the textbox and the listbox to know as little as possible about each other. In fact, the way the class is written, either of those controls could be replaced with something different or another control could be added to the container without having to change the code inside the individual controls. All the changes would take place at the container level. Creating the Position and Search methods gives us the most flexibility for future changes.

This container control behaves like Help's Index page, except for two things, one of which I think is better behavior. The first issue is that the pointer in the list doesn't move when the user adds a space to the search string. That is, in the example in Figure 2, if the user types a space, the list highlight doesn't move from "Mary" to "Mary Margaret". This is because the search string gets trimmed in Search. Trimming is necessary to do partial string matching. You could keep track of the string length and make sure only blanks typed by the user end up in the search string, but the code to do so is complex and failing to move the pointer on a trailing blank seems fairly unimportant.

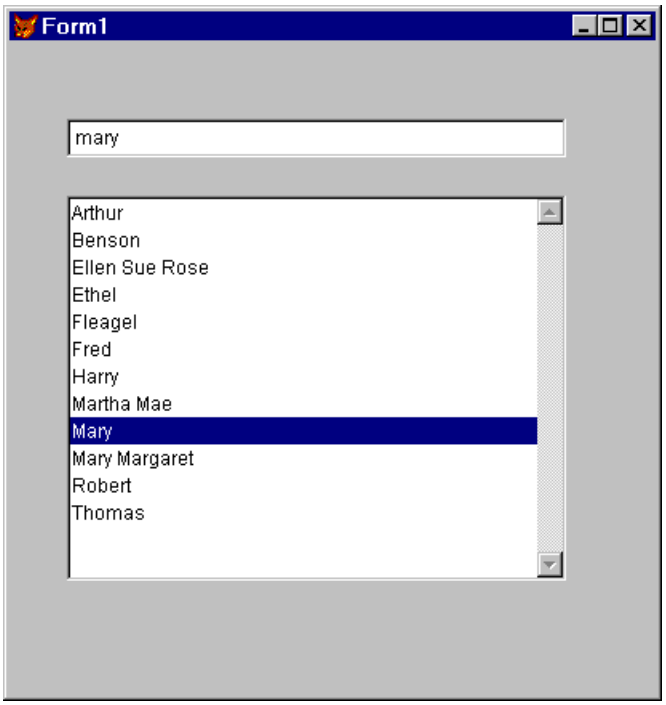


Figure 2. Using the incremental search container - The highlight in the list always is on the first entry that matches the text shown in the texbox. However, adding a space in the textbox doesn't move the highlight.

I think the second difference is actually an improvement. In Help's Index page, if the user types any characters at all in the listbox, the highlight moves to the first item in the list. In our container, when the user types, the highlight moves to the matching item and the textbox is properly updated.

You'll find the class on this month's Companion Resource Disk. There are a number of things you'll probably want to do to improve this class. First is to add some code to make sure that a RowSource for the list has been specified and that the RowSource is ordered. In addition, you may want to set the container's Resize event so that it resizes the contained controls. (See Ask Advisor in the March 1996 issue to learn how to resize the controls in a container.) You can probably think of other enhancements as well.

–Tamar