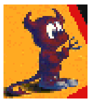
UnixWorld Online: Tutorial: Article No. 009: Part 2



The vi/ex Editor, Part 2: Line-Mode Addresses

By Walter Alan Zintz.

Questions regarding this article should be directed to the author at walter@ccnet.com.

[*Editor's note*: We'll surround text to be typed with actual double quotes, like: enter: "vi file1". Of course, don't type the quotes unless so instructed. We'll use ‘‘balanced quotes’’, constructed using pairs of backward and forward quotes, for all other purposes.]

Whenever you want to give an editor command that will operate on text that's already in the file you're editing--to delete some text, change lower-case letters to capitals, write to a file, etcetera--the editor needs to know what part of the file to go to work on. A few commands have their addresses built in, and most line-mode commands have default addresses that the editor will use if you don't give an address, but that still leaves a lot of occasions where you need to know how to give the editor an address and what address to give.

Many line-mode commands are almost identical to corresponding commands in visual mode; many more do similar things in different ways. Most of the benefit of these duplicative command sets comes from the totally-different addressing styles of line and visual modes. The differing address concepts mean that an edit that would be difficult or impossible to do with one mode's available addresses can be a piece of cake with an address form found in the other mode.

Since I mention ‘‘line mode’’ so often, you may wonder whether there really is a separate mode for line editing. There surely is--instead of filling your screen with text from the file you're editing, this mode gives you a colon (:) prompt for your line mode commands, and prints only an occasional line from the file on your screen. The feel of this mode is very much like giving UNIX commands from your shell prompt. Few people work in line mode these days, largely because you can give most line-mode commands from visual mode, but you can't give any visual-mode commands while you are in line mode. Or perhaps they just prefer the comfortable WYSIWYG feeling of seeing the text on screen, with changes appearing as they are made.

But there are times when you will need to work temporarily in line mode. To get to line mode when you first launch the editor, invoke it by typing "ex" instead of "vi". To go to line mode when you are already in the editor's visual mode, enter "Q". To get back to visual mode, type "vi" followed by a carriage return.

Wondering why I didn't put a colon in front of that command to return to visual mode, which is obviously a line-mode command? Because you don't need to type that colon when you're giving a command from within line mode. It may even be harmful; the rule is that if you type a colon at the start of a command from within line mode, there must be nothing between the colon and the command name or abbreviation. Not an address, not even a space, nothing at all.

So from this point on, I will display line-mode commands without an initial colon, because you now know enough to type that colon only if you're working in visual mode. And I'll leave off the tag at the end of a line-mode command that reminds you to finish with a carriage return because you now realize that any line-mode command, given from either line or visual mode, has to end with a carriage return.

Some of you may ask why I show line-mode command lines in long-winded form, with spelled-out command names and lots of whitespace instead of using abbreviations. For instance, the two command lines:

```
global /^/ move 0
g/^/m0
```

are identical in their effect, and the second is surely faster to type, so why do I show the first form? Because the long version is much easier to follow when I'm demonstrating a new concept, and almost everything here will be new to at least some of you. And it's a good idea to get to know the long forms, because you'll soon be learning to write editor scripts, and those scripts would be as cryptic as APL to future maintenance programmers if you wrote them in terse style. When I go over the roster of line-mode commands, I'll tell you both the long name and one or two short names for each.

Line-Mode Addressing

A SINGLE ADDRESS is often all you need with a line-mode command. One address refers to just one line, which tells a command like `delete` or `substitute` to operate on that one line only. A command like `insert` or `read`, which puts something immediately before or after a particular line, has no use for more than one address.

A search pattern, as discussed in the first installment of this tutorial, is always an acceptable line-mode address. You put the address at the start of the command line, before the command name (but after the initial colon if you are giving the command from visual mode), so:

```
?the [cC]at? delete
```

will erase the last previous line that contains the string ‘‘the cat’’ or ‘‘the Cat’’, while:

```
/^GLOSSARY$/ read gloss.book
```

puts the contents of the file ‘‘gloss.book’’ right after the next line in the file you're editing that contains only the word ‘‘GLOSSARY’’.

There are two shorthand forms for reusing search patterns as addresses. Typing "??" or "//" tells the editor to use the last search pattern you used previously, and your choice of "??" or "//" will set the direction of the search, overriding the direction you chose the previous time you used that search pattern. That is, if you type:

```
?the cat? yank
// delete
?? print
```

the second command will search forward, to remove the last previous line containing the string ‘‘the cat’’, even though your original use of that pattern was in a backward search. The third command will search backward to find the line to print, which (by coincidence) is the direction of the original search.

But the search pattern that those preceding abbreviations reuse may not be a pattern you used to search for a line. If you ran a `substitute` command after any pattern searches for lines, then the pattern you gave the `substitute` command to tell it what text to take out of the line is the pattern that will be reused. This is so even if your `substitute` command began with a search pattern to specify the line on which the substitution was to be performed--the search to find the pattern to be replaced within the line was run after the first search pattern had found the line to operate on, so the search within the line was the last pattern search run. So if you were to type:

```
/the cat/ substitute /in the hat/on the mat
?? delete
```

the second command would, in this case, delete the last previous line containing ‘‘in the hat’’. To be sure that the pattern that gets reused is the last one used to find a line, use the abbreviations "\?" and "\/" to search backward and forward, respectively. In all other respects these work just as typing "??" and "//" do.

A LINE NUMBER is also a valid line-mode address. The editor automatically numbers each line in the file consecutively, and this numbering is dynamic--that is, whenever you add or delete lines somewhere, the editor renumbers all the lines following the insertion or deletion point. So if you change some text on line 46 in your file, and then delete lines 11 and 12, the line with the text you changed is now line 44. And if you then add ten new lines after line 17, the line with your changed text on it now automatically becomes line 54.

There is never a gap or an overlap in the line number sequence, so the *n*th line in the file is always line number *n*; that is, the 7th line is always line number 7, and so on. (There are several ways to display these line numbers, which I will expound in a later tutorial installment.) To delete the 153rd line in your file, just type:

```
153 delete
```

You don’t use any delimiters around a line number, or around any other address except a search pattern.

There are two symbolic line numbers and one fictional one that can be used in line-mode addresses. As long as there are any lines in the buffer (that is, you haven’t specified a not-yet-existent file to edit and failed to enter any text so far), the editor regards you as being ‘on’ one of them, usually the last line affected by your latest command. A period or dot (`.`) is the symbolic address for this line. The last line in the file also has a symbolic address: the dollar sign (**`$`**$). So if you should type:

```
. write >> goodlines
```

```
$ delete
```

the first command would append a copy of just the line you are on now to a file named ‘‘goodlines’’, while the second would delete the last line in the file you are editing.

A few commands put text immediately after the line address you give: the `append` command is one of them. In order to let them put their text at the very start of a file (if that is where you want it), these commands can take the fictitious line number zero (0) as their address. So, if you want to type some text that will appear ahead of anything already in the file, you can do it with either of these command lines:

```
1 insert
0 append
```

(Note, though, that `insert` and `append` are among the few line-mode commands that cannot be run from visual mode by starting with a colon, because they occupy more than one line including the text to be put in.)

WRITING YOUR OWN LINE ADDRESSES is possible, too. You can attach lower-case letters to lines as line addresses, and change the attachments whenever you like. You can even use a special address that is automatically attached to the last line you jumped off from.

There are ways to mark a particular line with a lower-case letter of the alphabet, and those ways differ between line and visual modes. I’ll be explaining all these ways in later installments of this tutorial. But once a line is marked, the line-mode address that refers to that line is just the single-quote character followed immediately by the lower-case letter with which the line was marked. So typing:

```
'b print
```

will display on the screen whatever line you have previously marked with the letter b, no matter where the line is in relation to where you are when you give the command. No need to tell the editor whether to search forward or backward; there can be only one line at a time marked with any one letter, and the editor will find that line regardless.

The editor does some line marking on its own, too. Whenever you move from one line to another by a non-relative address, the editor marks the line you just left. (A non-relative address is one that isn’t a known number of lines from where you were.) So:

```
$
/the cat/
358
?glossary? +7
'b
```

are all non-relative addresses, and if you give any one of them, the editor will mark the line you are leaving for future reference. Then you can return to that line just by typing two successive single quotes:

```
''
```

as a line-mode address. In theory, you can use this address with any line-mode command. But it is so difficult to know for sure when you left a line via a non-relative address that this address form is best saved for going back to where you were when a mistake moves you far away, at least until you're a wizard with this editor.

MODIFYING ANY OF THESE ADDRESSES is possible, and there are two ways to do this. The simpler way is to offset the address a certain number of lines forward or backward with plus (+) or minus (−) signs. The rule is that each plus sign following an address tells the editor to go one line farther forward in the file than the basic address, while each minus sign means a line backward. So these three addresses all refer to the same line:

```
35
37 --
30 +++++
```

Not that you're likely to want to modify line-number addresses with counts, unless you're weak in arithmetic and want the editor to do the adding and subtracting for you. But the count offsets will work with any line-mode addresses, and are most often used with search patterns. In any event, there is a shorthand for these counts, too. A plus or minus sign immediately followed by a number (single or multiple digits) is equivalent to a string of plus or minus signs equal to that number, so that these two addresses are the same:

```
/^register long/ ++++
/^register long/ +4
```

Take note that the ''4'' in the second example does not mean ''line number 4'', as it would if it appeared by itself as an address. After a plus or minus sign, a number is a count forward or backward from where the primary address lands (or if there is no primary address before the count, from the line you are on when you run the command).

Note also that this is one of the few places in line-mode commands where you may not insert a blank space. The number must start in the very next character position after the plus or minus sign. If you violate this rule, the editor will uncomplainingly operate on some line that definitely is not the line you expected.

The second style of address modifier is used where you want to do a search that's complex. Let's say you want to go forward in the file to delete a line that starts with ''WARNING!'', but not the first such line the editor would encounter; you want the second instance. Either of these command lines will do it:

```
/^WARNING!/ ; /^WARNING!/ delete
/^WARNING!/ ; // delete
```

A semicolon (`;`) between two search patterns tells the editor to find the location of the first pattern in the usual way, then start searching from that location for the second pattern. In this case, the first search pattern turned up the first instance of a line starting with ''WARNING!'', and the second search pattern led the editor on to the second instance.

A very significant point here is that this combination of two search patterns, either of which could be a line address in itself, does not tell the editor to delete two lines. The semicolon means that the first pattern is merely a way station, and that the single line found by the second search pattern is the only line to be deleted. In brief, what looks like addresses for two lines is actually only an address for one. (This is not what the official documentation for this editor says, but the documentation is just plain wrong on this point.)

But that’s just the start of what you can do. You are not restricted to just two addresses. I’ve used up to ten of them, all separated by semicolons, to reach one specific line. As an example:

```
?^Chapter 3$? ; /^Bibliography$/ ; /^Spinoza/ ; /Monads/
```

will bring me to the title line of Spinoza’s first work with ‘‘Monads’’ in the title, in the bibliography for Chapter 3.

Nor are you limited to search pattern addresses when putting together a semicolon-separated address string. If you want to reach the first line following line 462 that contains the word ‘‘union’’, typing:

```
462 ; /\<union\>/
```

will bring you there. And any of the addresses can take numerical offsets, so:

```
462 +137 ; /register int/ ---
```

is also a legitimate address string.

But there are two unfortunate limitations on using semicolon-separated address strings. The lesser problem is that such a string can use ‘‘line zero’’ as an address only if the command following the address string could take line zero by itself as its address. That is, you can’t even start at line zero and then proceed elsewhere with additional addresses, unless the command can operate from line zero. So:

```
0 ; /Spinoza/ +++ ; /Kant/ delete
```

which looks like a reasonable way to be sure your search will find the very first ‘‘Spinoza’’ in your file, will actually fail with an error message about an illegal address.

The larger misfortune is that each address in a semicolon-separated string must be farther down in the file than the one that precedes it. (This means the actual location found, after applying any plus-sign or minus-sign offset.) You cannot move backward within the series of way points.

But that does not mean that you cannot use a backward search pattern within the string. The first address can be a backward search, of course. And a subsequent address can search backward if you are certain that the line it finds will actually be more forward in the file. For example, you may know that a certain backward search will wrap around to the bottom end of the file before it finds a match. A common example would be:

```
1 ; ?Spinoza? ; /Hegel/ yank
```

Beginning a backward search from the first line in the file means that the search must start with the last line in the file due to wraparound, which guarantees that the search will yank the ‘‘Hegel’’ line that follows the vary last ‘‘Spinoza’’ line in your file.

Also, you can use a plus-sign offset after a backward search when you are certain that the line finally found after the offset is applied will be farther down in the file than the preceding way point had been. Thus, if I want to find the first mention of Hegel in Chapter 8 that is at least 120 lines after the last mention of him in Chapter 7, I can type:

```
/^Chapter 8$/ ; ?Hegel? +119 ; //
```

If a command with this address fails and gives an error message about a bad address, I’ll know that the last mention of Hegel in Chapter 7 is more than 120 lines before the end of the chapter, so the very first mention of his name in Chapter 8 is what I’m looking for. In that case, the address:

```
/^Chapter 8$/ ; /Hegel/
```

is all that my command needs.

The situation with forward searches inside a semicolon-separated address string is a mirror image of what I’ve just said. A forward search can take a minus-sign offset if you know that the offset is small enough that the line found will be further down than the last way point. But a forward search will fail, even with no offset or a plus-sign offset, if wraparound makes it find a line earlier in the file than the way point from which it began.

Addressing a Section of Text

TWO ADDRESSES CAN ALSO STAND FOR A RANGE OF LINES. When two addresses are separated by a comma rather than a semicolon, the meaning changes radically. (What a difference a dot makes!)

Often you will want a line-mode command to act on a series of successive lines. For example, you may want to move a stretch of text from one place to another. To do this, you give the address of the first line you want the command to act on, followed by the last line it should act on, and separate the two addresses with a comma. So, the command:

```
14 , 17 delete
```

will delete line 14 and line 15 and line 16 and line 17. You can see that putting more than two addresses in a comma-separated address string would be pointless. The line mode of this editor is discreet if you ignore this and string together three or more addresses with comma separation: it uses the first two addresses and discards the rest.

Any line-mode addresses may be used with a comma. All of the following combinations make sense:

```
'd , /^struct/
```

```
257 , .
?^Chapter 9$? , $
```

The first address combination would cause the command that follows it to operate on the section starting with the line you have previously marked ‘‘d’’ and ending with the next forward line that begins with ‘‘struct’’, inclusive. The second combination covers line 257 through the line you are on now. The third goes backward to include the previous line containing only ‘‘Chapter 9’’, and forward to include the very last line in your file; plus all the lines in between, of course.

There are limitations on this technique, too. The primary one is that the address after the comma (after any offsets, of course) cannot be earlier in the file than the address before the comma. That is, the range of lines must run forward from the first address to the second address. So the command:

```
57 , 188 delete
```

is just fine, while the similar-looking command:

```
188 , 57 delete
```

will only produce an error message. (But if the two addresses happen to evaluate to the same line, there is no problem. The command will silently operate on the one line you’ve specified.)

As you work up to more sophisticated line-mode addresses, you may get unexpected error messages about the second address being prior to first address, when you don’t see how you could have anticipated that the addresses would evaluate that way. That’s no disgrace, and the solution is simple. After you’ve looked over the addresses you used, and you’re certain that they are the ones you want, just type the command in again with the two addresses in reverse order. That is, if:

```
642 , /in Table 23/ delete
```

has failed, giving an error message that the lines are in the wrong order, then:

```
/in Table 23/ , 642 delete
```

will solve that problem.

The last limitation is that when you use search patterns on both sides of a comma, the second search starts from the current line just as the first search did; it does not start from the line that the first search found. There’s a way around that, though, that involves using one or more semicolons along with a comma.

A semicolon-separated address string can be used anywhere in line mode that you would use a single address. One very useful technique is to use these address strings on one or both sides of a comma, to indicate a range of lines to be affected. Remember that an address string separated by semicolons is the address of just one line, so this one line can be the start or the end of a range of lines. For example, in:

```
/^INDEX$/ ; /^Xerxes/ , $ write tailfile
```

```
?^PREFACE$? ; /^My 7th point/ , ?^PREFACE$? ; /^In summary/ -- delete
```

the first command would write the latter part of the index to a new file, while the second could be used to remove a section of a book's preface.

And that brings up the solution to our previous obstacle; the second search's starting point. If you want the search after the comma to begin from the point the first search found, use the first search pattern followed by a semicolon as the start of your after-the-comma search string, as in either of:

```
?Stradivarius? , ?Stradivarius? ; /Guarnerius/
?Stradivarius? , ?? ; /Guarnerius/
```

In view of the rules about not going backward in line-mode address strings, I'd better clarify the way these limitations work when you combine semicolon and comma separation, as in these two examples. All but the first of the way points in each semicolon-separated string must be in the forward direction, of course, but the start of the second semicolon-separated string may be prior to any of the addresses in the first such string, that is, the one-way meter resets itself at the comma point. And using semicolon-separated strings on both sides of a comma only requires that the final landing point of the second semicolon-separated string not be earlier in the file than the final landing point of the first; the relative locations of the way points don't matter to the comma. To clarify this, consider a couple of odd-looking, and useless, but very lucid examples. The combination:

```
125 ; 176 ; 221 , 32 ; 67 ; 240
```

looks invalid due to the backward jump from line 221 to line 32, but is actually a perfectly good address. The back jump comes right after the comma, where it is all right. But:

```
125 ; 176 ; 221 , 32 ; 67 ; 218
```

will produce an error message, because the final landing point of the first semicolon-separated string, line 221, falls later in the file than the final landing point of the second semicolon-separated string, line 218.

Now, a note about default addresses. I've already mentioned that most line-mode commands that can take an address have a ''default'' address built in, which tells the editor where to run the command if you don't give an address with it. Each command has its own default address, which may be the current line, the current line plus the one following, the last line of the file, or the entire file.

The comma separator has default addresses of its own. They are the same regardless of what command is being used, and they override any command's own default address. If you put a comma before a command and don't put an address before the comma, by default the address there is the current line. In the same way, if you leave out the address after the comma, the default there is also the current line. You can even leave out the address in both places and use the current-line default in both: that means the implied address is ''from the current line to the current line'', which makes the current line the only line the command will operate on. So every one of the following command lines:

```
.     write >> goodlines
```

```
. , . write >> goodlines
  , . write >> goodlines
. ,   write >> goodlines
  ,   write >> goodlines
```

will do exactly the same thing: append a copy of just the current line in the file you're editing to another file named ‘‘goodlines’’.

Finally, there is one special symbol that represents a comma-separated address combination. The percent sign (`%`) has the same meaning as `1,$` as a line-mode address combination. Both refer to the entire file.

Now You Try It

Before you try the complex aspects of line-mode addresses in actual editing situations, here are some problems you can build yourself up on. For each problem I've included a solution that will work fairly efficiently.

1. How can you tell the editor to delete the line that holds the very last instance of ‘‘EXPORT’’ in your file? The solution is straightforward once you know where to start searching.

2. Suppose you want to delete the very first line in the file with ‘‘EXPORT’’ on it, and that just might be line 1. You can't start the search from line zero because the delete command cannot take line 0 as an address. When you type the address string "$; /EXPORT/" to use wraparound, you get an error message asserting that the search pattern found a line prior to the line found by the ‘‘$’’ address that appeared first, which is what you'd expect. How can you tell the editor to find and delete this line? The solution requires just a bit of creativity.

3. If you use the address "?abc? , /xyz/", it includes the two lines the searches (for ‘‘abc’’ and ‘‘xyx’’) find, as well as all the lines between them. How would you specify that you want the affected lines to go up to, but not include, the lines the two searches find? In this case the solution is simpler than you might think.

Coming Up Next

The next installment of this tutorial will deal with the `global` commands--they're just too much to absorb right after the mind-numbing collection of address forms we've just gone through. And to give you more scope for using all these address forms, I'll also cover line-mode commands themselves, particularly the ones that have more capabilities than you suspect.

Edited by Becca Thomas / Online Editor / UnixWorld Online / editor@unixworld.com

Go To Content   Search

Last Modified: Thursday, 04-Apr-96 10:16:48 PST