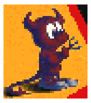
UnixWorld Online: Tutorial: Article No. 009



Part 1: Vi Editor Fundamentals

By Walter Alan Zintz.

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

[*Editor's Note:*. This article is a ‘‘work in progress’’ and will evolve over time. We'll announce each new addition.]

Why Vi?

A HEARTWARMING EDIT. Pity poor Hal, a corporate maintenance programmer. A large module of badly- broken, poorly-patched legacy code--the spaghetti variety--finally broke down completely yesterday, leaving one corporate division running at half speed. By dint of some inspired fixes during an all-nighter, Hal has the module up and running again this morning...but just as he's ready to go out for food that isn't from a vending machine, in walks the corporation's VP of IS, with a big surprise.

> ‘‘Nice work on that crash fix, Hal; but right now I need some formatted technical data about it, in a hurry. The Board of Directors' Information Systems Committee has called a rush meeting this morning to convince themselves they're on top of the problem. I'll be in the hotseat, and I need technical data I can put up on the video projector to keep them occupied.
>
> ‘‘They'll want me to discuss the logfile of errors that led up to the crash . . . yes, I know that's in `/oltp/err/m7`, but appending puts the latest report lines at the bottom of the file. Those suits aren't interested in what they think is ancient history, and they wouldn't be caught reading anything but a commuter train timetable from the bottom up, so you'll have to make a copy with the order of the lines reversed: what was the last line becomes the first

> line, what was the second to the last line is now line number two, and so on.
>
> ‘‘And let’s take a look at that logfile.
>
> ```
> 374a12  44872  130295/074457  nonabort
> 5982d34  971  130295/221938  nonabort
> 853f7  2184  140295/102309  abort
>  ...
> ```
>
> Hmmm. Explaining the second column to them would be advertising the fact that we knew this failure was just waiting for a chance to happen. So while you’re at it, go through and erase all but the first and last digits of each number in column two.
>
> ‘‘Oh, and when they get tired of that they’ll want to scrutinize the Lint report. Last month I told them that our Lint substitute was the greatest thing since Marilyn Monroe, so now they’ll want me to tell them why the messages it still generates on this module aren’t real hazards. Just run Lint over the revamped module; then combine the Lint output with a copy of the source file by taking each message line like:
>
> ```
> Line 257: obsolete operator +=
> ```
>
> and putting the significant part at the end of the source line it refers to. And put a separator, like `XXX`, between the source line and the message so I can page through quickly. Nothing like a hefty dose of source code they can’t begin to fathom to make the meeting break up early.
>
> ‘‘And get right on this. The meeting starts in 35 minutes.’’

Our VP walks away inwardly smiling, thinking he’s getting out of detailed explanations and putting all the blame on an underling, just by demanding more editing than anyone could do in the time available. ‘‘I’ll tell the Information Systems Committee that I made it perfectly clear to the programmer that we needed this at 9:30, but when I asked him for it a minute ago he said it wasn’t finished and he wasn’t sure when it would be. Then I’ll remark that those programmers just can’t understand that keeping management informed is every bit as important as writing code!’’

But Hal has a secret weapon against this squeeze play: an expert knowledge of the Vi editor.

Reversing the order of the lines in a file is a piece of cake with this editor. The eight keystrokes in:

:g/^/m0 ***(ret)***

will do it. Taking the digits out of the middle of the second column throughout the file also requires just one command line:

:%s/^\([^]* [0-9]\)[0-9]*\([0-9] \)/\1\2 ***(ret)***

And integrating the Lint messages into a copy of the source code? Even that can be automated with the Vi editor. The editor command:

:%s/Line \([0-9][0-9]*\): \(.*\)/\1s;$; XXX \2 ***(ret)***

will turn that file of Lint messages into an editor script, and running that script on a copy of the source file will mark it up as requested.

Rather than being portrayed as a bungler, Hal can have it all ready in a couple of minutes, just by typing a few lines. He'll even have time to guard against vice-presidential prevarication, by disappearing into the coffee shop across the street and reappearing just as the meeting is getting started, to tell the VP (and everyone else in earshot), ''Those files you wanted are in slash-temp-slash-hal''.

THE PLAN OF THIS ONGOING TUTORIAL. I'm writing here for editor users who have some fluency in Vi/Ex at the surface level. That is, you know how to do the ordinary things that are belabored in all the ''Introducing Vi'' books on the market, but rarely venture beyond that level.

This tutorial series will explore a lot of other capabilities that hardly anyone knows are in Vi/Ex. That includes quite a few tricks that may be built on editor functions we all use every day, but which nonetheless are not obvious--for instance, telling the global command to mark every line it encounters. I'll also be clarifying the real nature of the many misunderstood aspects of this editor.

To do all this, I'll be explaining things in more depth than you might think warranted at first. I'll also throw in examples wherever they seem helpful. And to save you readers from gross information overload, I'll write this tutorial in a large number of fairly small modules, to be put up on our website at a calm, reasonable pace.

The Editor's Basic Concepts

To get a real grasp on this editor's power, you need to know the basic ideas embodied in it, and a few fundamental building blocks that are used throughout its many functions.

One cause of editor misuse is that most users, even experienced ones, don't really know what the editor is good at and what it's not capable of. Here's a quick rundown on its capabilities.

First, it's strictly a general-purpose editor. It doesn't format the text; it doesn't have the handholding of a word processor; it doesn't have built-in special facilities for editing binaries, graphics, tables, outlines, or any programming language except Lisp.

It's two editors in one. Visual mode is a better full-screen editor than most, and it runs faster than those rivals that have a larger bag of screen-editing commands. Line editing mode dwarfs the ''global search and replace'' facilities found in word processors and simple screen editors; its only rivals are non-visual editors like Sed where you must know in advance exactly what you want to do. But in the Vi/Ex editor, the two sides are very closely linked, giving the editor a combination punch that no other editor I've tried can rival.

Finally, this editor is at its best when used by people who have taken the trouble to learn it thoroughly. It's too capable to be learned well in an hour or two, and too idiosyncratic to be mastered in a week, and yet the power really is in it, for the few who care to delve into it. A large part of that power requires custom-programming the editor: that's not easy or straightforward, but what can be done by the skillful user goes beyond the direct programmability of any editor except (possibly) Emacs.

Search Patterns

In quite a few functions of this editor, you can use string-pattern searching to say where something is to be done or how far some effect is to extend. These search patterns are a good example of an editor function that is very much in the Unix style, but not exactly the same in detail as search patterns in any other Unix utility.

Search patterns function in both line editing and visual editing modes, and the work the same way in both, with just a few exceptions. But how you tell the editor you're typing in a search pattern will vary with the circumstances.

SEARCHING FROM WHERE YOU ARE NOW. The more common use for search patterns is to go to some new place in the file, or make some editing change that will extend from your present position to the place the pattern search finds. (In line editing mode it's also possible to have an action take place from one pattern's location to where another pattern is found, but both searches still start from your present location.)

If you want to search forward in the file from your present location (toward the end of the file), precede the search pattern with a slash (**/**) character, and type another to end the pattern. So if you want to move forward to the next instance of the string ''j++'' in your file, typing:

/j++/*(ret)*

will do it. And so will:

/j++*(ret)*

When there is nothing between the pattern and the RETURN key, the RETURN itself will indicate the end of the search pattern, so the second slash is not necessary. And if you are in visual mode, the ESCAPE key works as well as RETURN does for ending search input, so

/j++*(esc)*

is yet another way to make the same request from visual mode.

To search backward (toward the start of the file), begin and end with a question mark instead of a slash.

The same rules of abbreviation apply to backward searches, so

?j++?*(ret)*
?j++*(ret)*
?j++*(esc)*

are all ways to head backward in the file to the same pattern.

Either way, you've expressed both your request for a pattern search and the direction the search is to take in just one keystroke. But don't assume that if you search backward, any matching pattern the editor finds will be above your present position in the file, and vice versa if you search forward. The editor looks there first, certainly, but if it gets to the top or bottom line of the file and hasn't found a match yet, it wraps around to the other end of the file and continues the search in the same direction. That is, if you used a question mark to order a backward search and the editor searches all the way through the top line of the file without finding a match, it will go on to search the bottom line next, then the second-to-the-bottom line, and so on until (if necessary) it gets back to the point where the search started. Or if you were searching forward and the editor found no match up through the very last line of the file, it would next search the first line, then the second line, etcetera.

If you don't want searches to go past either end of the file, you'll need to type in a line mode command:

:set nowrapscan ***(ret)***

This will disable the wraparound searching during the present session in the editor. If you want to restore the wraparound searching mechanism before you leave the editor, typing

:set wrapscan ***(ret)***

will do it, and you can turn this on and off as often as you like.

THE FIND-THEM-ALL SEARCH. Up to now, I've been considering searches that find just one instance of the pattern; the one closest to your current location in the file, in the direction you chose for the search. But there is another style of search, used primarily by certain line editing mode commands, such as `global` and `substitute`. This search finds every line in the file (or in a selected part of the file) that contains the pattern and operates on them all.

Don't get confused when using the `global` and `substitute` commands. You'll often use both styles of search pattern in one command line. But the find-one-instance pattern or patterns will go before the command name or abbreviation, while the find-them-all pattern will come just behind it. For example, in the command:

:?Chapter 10?,/The End/substitute/cat/dog/g ***(ret)***

the first two patterns refer to the preceding line closest to the current line that contains the string ''Chapter 10'' and the closest following line containing the string ''The End''. Note that each address finds only one line. Combined with the intervening comma, they indicate that the `substitute` command is to operate on those two lines and all the lines in between them. But the patterns immediately after the `substitute` command itself tell the command to find every instance of the string ''cat'' withing that range of lines and replace it with the string ''dog''.

Aside from the difference in meaning, the two styles also have different standards for the delimiters that mark pattern beginnings and (sometimes) endings. With a find-them-all pattern, there's no need to indicate whether to search forward or backward. Thus, you aren't limited to slash and question mark as your pattern delimiters. Almost any punctuation mark will do, because the editor takes note of the first punctuation mark to appear after the command name, and regards it as the delimiter in that instance. So

:?Chapter 10?,/The End/substitute;cat;dog;g ***(ret)***
:?Chapter 10?,/The End/substitute+cat+dog+g ***(ret)***
:?Chapter 10?,/The End/substitute{cat{dog{g ***(ret)***

are all equivalent to the substitution command above. (It is a good idea to avoid using punctuation characters that might have a meaning in the command, such as an exclamation point, which often appears as a switch at the end of a command name.)

The benefit of this liberty comes when the slash mark will appear as itself in the search pattern. For example, suppose our substitution command above was to find each pair of consecutive slash marks in the text, and separate them with a hyphen--that is, change `//` to `/-/`. Obviously,

```
:?Chapter 10?,/The End/substitute/////-//g(ret)
```

won't work; the command will only regard the first three slashes as delimiters, and everything after that as extraneous characters at the end of the command. This can be solved by backslashing:

```
:?Chapter 10?,/The End/substitute/\/\//\/-\//g(ret)
```

but this is even harder to type correctly than the first attempt was. But with another punctuation mark as the separator

```
:?Chapter 10?,/The End/substitute;//;/-/;g(ret)
```

the typing is easy and the final command is readable.

SIMPLE SEARCH PATTERNS. The simplest search pattern is just a string of characters you want the editor to find, exactly as you've typed them in. For instance: ''the cat''. But, already there are several caveats:

1. This search finds a string of characters, which may or may not be words by themselves. That is, it may find its target in the middle of the phrase ''we fed *the cat* boiled chicken'', or in the middle of ''we sailed a li*the cat*amaran down the coast''. It's all a matter of which it encounters first.
2. Whether the search calls ''The Cat'' a match or not depends on how you've set an editor variable named *ignorecase*. If you've left that variable in its default setting, the capitalized version will not match. If you want a capital letter to match its lower-case equivalent, and vice versa, type in the line mode command

   ```
   :set ignorecase(ret)
   ```

   To resume letting caps match only caps and vice versa, type

   ```
   :set noignorecase(ret)
   ```

3. The search absolutely will not find a match where ''the'' occurs at the end of one line and ''cat'' is at the start of the next line:

   ```
   and with Michael's careful help, we prodded the
   cat back into its cage.  Next afternoon several
   ```

   It makes no difference whether there is or isn't a space character between one of the words and the linebreak. Finding a pattern that may break across a line ending is a practically impossible task with this line-oriented editor.
4. Where the search starts depends on which editor mode you're using. A search in visual mode starts with the character next to the cursor. In line mode, the search starts with the line adjacent to the current line.

METACHARACTERS. Then there are search metacharacters or ''wild cards'': characters that represent something other than themselves in the search. As an example, the metacharacters `.` and `*` in

```
/Then .ed paid me $50*!/(ret)
```

could cause the pattern to match any of:

```
Then Ted paid me $5!
Then Red paid me $5000!
Then Ned paid me $50!
```

or a myriad of other strings. Metacharacters are what give search patterns their real power, but they need to be well understood.

To understand these, you must know the varied uses of the backslash (\) metacharacter in turning the ‘‘wild card’’ value of metacharacters on and off.

In many cases, the meta value of the metacharacter is on whenever the character appears in a search pattern unless it is preceded by a backslash; when the backslash is ahead of it the meta value is turned off and the character simply represents itself. As an example, the backslash is a metacharacter by itself, even if it precedes a character that never has a meta value. The only way to put an actual backslash in your search pattern is to precede it with another backslash to remove its meta value. That is, to search for the pattern ‘‘a\b’’, type

```
/a\\b/(ret)
```

as your search pattern. If you type

```
/a\b/(ret)
```

the backslash will be interpreted as a metacharacter without any effect (since the letter b is never a metacharacter) and your search pattern will find the string ‘‘ab’’.

Less-often-used metacharacters are used in exactly the opposite way. This sort of character represents only itself when it appears by itself. You must use a preceding backslash to turn the meta value on. For example, in

```
/\<cat/
```

the left angle bracket (<) is a metacharacter; in

```
/<cat/
```

it only represents itself. These special metacharacters are pointed out in the list below.

Finally there is a third class, the most difficult to keep track of. Usually these metacharacters have their meta values on in search patterns, and must be backslashed to make them represent just themselves: like our first example, the backslash character itself. But if you’ve changed the default value of an editor variable named *magic* to turn it off, they work oppositely--you then must backslash them to turn their meta value on: like our second example, the left angle bracket. (Not that you are are likely to have any reason to turn *magic* off.) These oddities are also noted in the list below.

And don’t forget the punctuation character that starts and ends your search pattern, whether it is slash or question mark or something else. Whatever it is, if it is also to appear as a character in the pattern you are searching for, you’ll have to backslash it there to prevent the editor thinking it is the end of the pattern.

TABLE OF SEARCH PATTERN METACHARACTERS

`.`

A period in a search pattern matches any single character, whether a letter of the alphabet (upper or lower case), a digit, a punctuation mark, in fact, any ASCII character except the newline. So to find ‘‘default value’’ when it might be spelled ‘‘default-value’’ or ‘‘default/value’’ or ‘‘default_value’’, etcetera, use **`/default.value/`** as your search pattern. When the editor variable *magic* is turned off, you must backslash the period to give it its meta value.

`*`

An asterisk, plus the character that precedes it, match any length string (even zero length) of the character that precedes the asterisk. So the search string **`/ab*c/`** would match ‘‘ac’’ or ‘‘abc’’ or ‘‘abbc’’ or ‘‘abbbc’’, and so on. (To find a string with at least one ‘‘b’’ in it, use **`/abb*c/`** as your search string.) When the asterisk follows another metacharacter, the two match any length string of characters that the metacharacter matches. That means that **`/a.*b/`** will find ‘‘a’’ followed by ‘‘b’’ with anything (or nothing) between them. When the editor variable *magic* is turned off, you must backslash the asterisk to give it its meta value.

`^`

A circumflex as the first character in a search pattern means that a match will be found only if the matching string occurs at the start of a line of text. It doesn’t represent any character at the start of the line, of course, and a circumflex anywhere in a search pattern except as the first character will have no meta value. So **`/^cat/`** will find ‘‘cat’’, but only at the start of a line, while **`/cat^/`** will find ‘‘cat^’’ anywhere in a line.

`$`

A dollar sign as the last character in a search pattern means the match must occur at the end of a line of text. Otherwise it’s the same as circumflex, above.

`\<`

At the start of a search pattern, a backslashed left angle bracket means the match can only occur at the start of a simple word; at any other position in a search pattern it is not a metacharacter. (In this editor, a ‘‘simple’’ word is either a string of one or more alphanumeric character(s) or a string of one or more non-alphanumeric, non-whitespace character(s), so ‘‘shouldn’t’’ contains three simple words.) Thus **`/\<cat/`** will find the last three characters in ‘‘the cat’’ or in ‘‘tom-cat’’, but not in ‘‘tomcat’’. To remove the meta value from the left angle bracket, remove the preceding backslash: **`/<cat/`** will find ‘‘<cat’’ regardless of what precedes it.

`\>`

At the end of a search pattern, a backslashed right angle bracket means the match can occur only at the end of a simple word. Otherwise the same as the left angle bracket, above.

`~`

The tilde represents the last string you put into a line by means of a line mode `substitute` command, regardless of whether you were in line mode then or ran it from visual mode by preceding it with a colon (‘‘`:`’’). For instance, if your last line mode substitution command was **`s/dog/cat/`** then a **`/the ~/`** search pattern will find ‘‘the cat’’. But the input string of a `substitute` command can use metacharacters of its own, and if your last use involved any of those metacharacters then a tilde in your search pattern will give you either an error message or a match that is not what you expected. When the editor variable *magic* is turned off, you must backslash the tilde to give it its meta value.

CHARACTER CLASSES. There is one metastring form (a ‘‘multicharacter metacharacter’’) used in search patterns. When several characters are enclosed within a set of brackets (`[]`), the group matches any one of the characters inside the brackets. That is, `/part [123]/` will match ‘‘part 1’’, ‘‘part 2’’ or ‘‘part 3’’, whichever the search comes to first. One frequent use for this feature is in finding a string that may or may not be capitalized, when the editor variable *ignorecase* is turned off (as it is by default). Typing `/[Cc]at/` will find either ‘‘Cat’’ or ‘‘cat’’, and `/[Cc][Aa][Tt]/` will find those or ‘‘CAT’’. (In case there was a slip of the shift key when ‘‘CAT’’ was typed in, the last pattern will even find ‘‘CaT’’, ‘‘CAt’’, etcetera.)

There’s more power (and some complication) in another feature of this metastring: there can be metacharacters inside it. Inside the brackets, a circumflex as the first character reverses the meaning. Now the metastring matches any one character that is NOT within the brackets. A `/^[^ ]/` search pattern finds a line that does not begin with a space character. (You’re so right if you think that the different meta values of the circumflex inside and outside the character class brackets is not one of the editor’s best points.) A circumflex that is not the first character inside the brackets represents just an actual circumflex.

A hyphen can be a metacharacter within the brackets, too. When it’s between two characters, and the first of the two other characters has a lower ASCII value than the second, it’s as if you’d typed in all of the characters in the ASCII collating sequence from the first to the second one, inclusive. So `/[0-9]%/` will find any numeral followed by the percent sign (`%`), just as `/[0123456789]%/` would. A `/[a-z]/` search pattern will match any lower-case letter, and `/[a-zA-Z]/` matches any letter, capital or lower case. These two internal metacharacters can be combined: `/[^A-Z]/` will find any character except a capital letter. A hyphen that is either the first or the last character inside the brackets has no meta value. When a character-hyphen-character string has a first character with a higher ASCII value than the last character, the hyphen and the two characters that surround it are all ignored by the pattern search, so `/[ABz-a]/` is the same as `/[AB]/`.

Backslashing character classes is complex. Within the brackets you must backslash a right bracket that’s part of the class; otherwise the editor will mistake it for the bracket that closes the class. Of course you must backslash a backslash that you want to be part of the class, and you can backslash a circumflex at the start or a hyphen between two characters if you want them in the class literally and don’t want to move them elsewhere in the construct. Elsewhere in a search pattern you will have to backslash a left bracket that you want to appear as itself, or else the editor will take it as your attempt to begin a character class. Finally, if *magic* is turned off, you’ll have to backslash a left bracket when you do want it to begin a character class.

Coming Up Next

In the second part of this tutorial, I’ll be following up on all this information about search patterns, by showing the right ways to combine them with other elements to generate command addresses. As a second part finale, I’ll show how to tap the enormous power of the command that looks like an address: the `global` command.

Edited by Becca Thomas / Online Editor / UnixWorld Online / beccat@wcmh.com

Last Modified: Monday, 18-Dec-95 11:22:08 PST