The Joy of Vim, Part Three

By Jerry Peek

This month, let's wrap up the series about a few of the powerful new features that make *vim Vi IMproved*: a visual text-selection mode, moving the cursor to places without text (yet), incrementing and decrementing numbers, storing editing commands in registers, handling binary files and different end-of-line styles, line breaking, and screen scrolling.

As before, if you don't have a *.vimrc* file in your home directory, you can enable the new features of *vim* features by typing…

```
:set nocompatible
```

… after you launch the editor.

Visual Mode: Seeing Before Doing

Both original *vi* and *vim* can modify an area of text by following a "change" command, such as `d` for "delete," with a motion command, like `w` for "word." So, `dw` deletes the next word, and the command `c/The dog` changes all text from the current cursor position up to but not including the next occurrence of `The dog`.

While expressive and useful, such commands work invisibly — that is, you can't see which text will be modified until you execute the command. Also, these commands only work on a contiguous string of characters; it's hard to, say, delete the third column of a table or the sixth through the tenth characters on each of the next five lines.

vim 's *visual mode* makes editing easy. It highlights text to be modified. Once you've selected some text, type a modification command.

There are several visual mode operations:

- `v` highlights text character-by-character. For example, type `v` and then type the motion command `/The dog` (and press Return). All characters from the current cursor position to the start of the string `The dog` are highlighted. If you then type `2w`, the next two words (`The dog`) are highlighted, too.

- `V` (uppercase "V") highlights text line-by-line. For instance, type `V`, then type `/The dog` (and Return) to highlight all of the text on the current line and all lines through the next line containing `The dog`. If you start with visual line mode (uppercase `V`), you can switch to visual character mode by typing lowercase `v`.

- To cancel visual mode without modifying the highlighted text, simply repeat the key for the current visual mode (`v` or `V`).

- Visual block mode, via Control-V, highlights a block of characters that doesn't necessarily include the start or end of any lines. (Original *vi* basically couldn't do this. You either selected a string of characters, or a series of lines, but they always included any newline characters along the way.) For example, you might want to copy ("yank") a couple of columns from your system's */etc/fstab* file and paste them into another file. To do that, put your cursor at the top-left corner of the block of text, type Control-V, then move the cursor down (`4j`) and right (`25l`), or whatever motion command you choose, until the text is highlighted, as in *Figure One*. Then type `y` to yank a copy of the text, switch to the other file, and type `p` to paste ("put") the copied text. (A newline is automatically added to the end of each pasted line.)

- After you select a block, your cursor is typically at the bottom of it. If you want to adjust the other end of the block, type `o` and then type other motion commands. Typing `o` again jumps back to the original place.

In block selection mode, you can go to the other side of the current line with `O` (uppercase "O") or the opposite corner of the block with `o`. `o` and `O` work differently in visual mode than in normal mode, where they open a new line below or above the cursor.

Virtual Editing

Unlike some editors, *vi* won't let you move the cursor to places where there is no text. For instance, if a line has ten characters, you can't move the cursor to the 11th character position unless you're using insert mode. This isn't convenient when, for instance, you're editing tables of text where some rows (lines) have a lot of fields and other rows have just a few fields. As you move the cursor up and down, it can jump back and forth instead of in a straight line.

The *vim* option `virtualedit` solves this problem. It lets you move the cursor to places where there's currently no text. *vim* won't actually add text to those places (or spaces or tabs before them) unless you switch to a mode like insert.

The setting of `virtualedit` is a comma-separated string containing one or more of `block`, which allows virtual editing in visual block mode, `insert`, allowing virtual editing during insert mode, and `all`, which allows virtual editing in all modes. If you're a longtime *vi* user, `:set virtualedit= block` may feel most natural.

Virtual editing is an example of one of the many new fea-

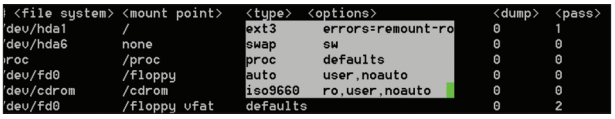

FIGURE ONE: Text selected with *vim's* visual block mode

tures added to *vim* you won't see without reading the documentation (or articles like this one).

Editing Numbers

Last month's column showed an example of using *vim*'s built-in scripting to insert item numbers automatically before list items. If your file already has numbers in it, though, you'll probably want to use another *vim* feature. The Control-A and Control-X commands, respectively, increment and decrement numbers.

If you'd like to experiment, type (or copy and paste) the following math exam questions into a file:

```
Q1: 010 octal equals 8 decimal? (T/F)
Q2: 0xa + 0x2 equals 11 decimal? (T/F)
```

Put the cursor on either character of `Q2` ("question 2") and press Control-A repeatedly. *vim* changes the string to `Q3`, `Q4`, and so on. Press Control-X to decrement the number.

Put the cursor on any character of `010` and type Control-X once. *vim* changes the number to `007` because it recognizes a number with a leading zero as octal. Type Control-A (or `u`, the undo command) to revert to `010`. In the same way, Control-X decrements the hex number `0xa` to `0x9`, and Control-A increments it to `0xb`.

These handy commands work when you're editing source code, tables, and most other numeric text. *vim* can also increment and decrement single letters like `a)`, `b)`, and so on. The `nrformats` option controls which formats *vim* recognizes; type `:help nrformats` for details.

Stored Editing With Registers

Both *vi* and *vim* have 26 string registers named `a` through `z`. You can use the registers to hold text with commands like `"ayt#`, which says, "Into register `a`, yank a copy of all text from here to the next character `#`. " Using an uppercase letter (here, `"A` instead of `"a`) appends text to the register instead of overwriting its previous contents. Later, you can paste the text with a command like `"aP`, which means, "Paste the text from register `a` before the cursor position."

vi (and *vim*) also have a clumsy way to use registers to store and execute commands. For instance, you can type the text `1Gd/URGENT` onto a line of the file you're editing, then move that text into register `x` by typing `"xdd` ("delete the current line into register `x`"). The ten characters `1Gd/URGENT`, plus the newline (important here!), are stored. Later, when you type the command `@x` ("execute the commands from register `x`"), the editor moves the cursor to the start of the first line (`1G`) and deletes all text until the string `URGENT` (`d/URGENT`, followed by a newline). You can repeat the commands by typing `@x` again or by typing `@@` ("repeat commands from the last register used").

That old system is handy for repetitive edits, but it's also hard to use. *vim* has made it so much easier with the new `q` ("record into register") command. To store a series of commands into a register:

1. Type `q` followed by the single-character register name. Use the lowercase letter (like `qx`) to replace previous register contents or the uppercase letter (`qX`) to append.

2. Type the commands you want to record. The commands also take effect as you type them.

3. Type `q` to stop recording.

So, let's say you type `qx`, type the commands `1Gd/URGENT` and press Return, then type `q` to stop recording. Later, you can re-execute the same commands by typing `@x`. It's similar to defining keymaps (explained in last month's column), but even easier to do.

"Binary" files, Multiple End-of-Line Styles

Old versions of *vi* could fail horribly when you tried to edit a file with lines that were "too long" (however long that was) or with certain non-text characters. Although *vim* wasn't really designed for editing non-text files, such as photos or executable programs, you can generally do it. Use the `-b` ("binary") option and see the help file (`:help binary`) for more information.

One of *vim*'s strengths is that it's a cross-platform editor. It runs natively on systems including *Linux*, *Unix*, *Microsoft Windows*, and more. Different operating systems have different ways to signal the end of a line in a text file, though. Unix and Linux text file lines end with `LF` (the linefeed character, 012 octal). *Macintosh* text file lines end with `CR` (carriage return, 015 octal). *DOS* and Windows text files end with `CR` followed by `LF`.

When you open an existing text file with *vim*, it guesses the file type. (It doesn't just use the format on your current operating system because you may have copied a file from a different system or may be editing over a network.) The possible file types are listed in *vim* 's option `fileformats`. To see the current setting, type `:set fileformats`:

```
:set fileformats
  fileformats=unix,dos
```

See Power, *pg. 62*

Power, *from pg. 17*

vim tries the formats in the order listed. End-of-line character (s) will be invisible and non-editable. If you *want* to see and/or edit these characters, though — for instance, if you want to strip some `CR` characters from a DOS-format file — you can force *vim* to re-open the file in a particular format by executing a command like the following from within *vim*:

```
:e! ++ff=unix %
```

(`:e!%` restarts editing with a fresh copy of the current file, whose name is available with the shortcut `%`. The `++ff=unix`, which can also be written `++fileformat=unix`, sets *vim*'s `fileformat` option to *unix*.) For more information, try `:help usr_23` and `:help file-formats`.

Scrolling and breaking

Original *vi* shows every character in every line of the file. If a line is longer than the window, extra characters are "wrapped" to the next line(s). If all of the characters in a line don't fit in the window, *vi* shows a series of `@` characters along the left margin instead. When you're editing a file with long lines, moving the cursor down one "line" can cause screen chaos.

vim will work that way if you'd like it to, but it can also work differently:

- `:set nowrap` displays as many characters from each line as possible. The rest aren't shown until you move the cursor past the left or right edge; that shows the next part of the current line. (To find the cursor's column number on the current line and the current line number, type CTRL-G.)
- `:set wrap` makes *vim* work like *vi*: lines are broken at the margin. Words will probably be broken across more than one line. (Lines aren't actually split. They're simply displayed on more than one line of the window.)
- `:set linebreak` breaks lines at word boundaries close to the margin. (Actually, it breaks lines at any of the characters listed in the `breakat` string.) This only changes the display of lines; it doesn't insert actual end-of-line characters into the middle of lines. (To actually break lines, see the options `wrapmargin`, `textwidth`, and `formatoptions`.)

A screen with multiple lines or parts of lines requires maneuvering to see the area you want.

`z` lets you scroll the screen so a particular line falls at the top (`z<CR>`), middle (`z.`), or bottom (`z-`). In *vim*, `z` has been extended to control both horizontal and vertical scrolling. You can scroll the view a half-screen left with `zH` (mnemonic: the `h` command moves the cursor left), half a screen right with `zL`. Type `:help scrolling` for all of the commands.

To always keep some text around the cursor, use the `scrolloff` and `sidescrolloff` options.

Contact Jerry Peek at http://www.jpeek.com/contact.html.