Why Use vim?, Part 2

By Jerry Peek

Last month's "Power Tools" column described two of the many things that make *vim* better than (but still compatible with) original *vi*. (You can get a list of more "most interesting" *vim* additions by typing `:help vim-additions` from within *vim*.) This month, let's dig into some of *vim*'s programmable features: improvements in key mapping, a scripting language, and built-in and user-defined functions. Put down your mouse and get ready to roll!

Be Incompatible

By default, *vim* acts as much like *vi* as it can, modulo the worst *vi* "bugs." However, some features require *vim* to act unlike *vi*. *vim* automatically "improves" itself if it finds a *.vimrc* file in your home directory. Otherwise, you can make *vim* act like *vim* by typing:

```
:set nocompatible
```

If you're following along from your keyboard, please run that command now.

Key Mapping Plus

Original *vi* lets you make simple "programs" by mapping one or more keys to run a sequence of commands. For instance, the following command maps the two-key sequence `\d` to surround the current word with double quote (`"`) characters:

```
:map \d i"^[Ea"^[
```

After defining this keymap, pressing `\` followed quickly by `d` inserts a double quote to the left of the cursor (`i"`), returns to command mode (`^[`, which represents an ESC character), moves to the end of the current word (`E`), appends a double quote to the right side of the cursor (`a"`), and returns to command mode (`^[`).

vim is easier to use.
You can even edit its command-line.

vim supports this syntax and improves upon it in two ways. Old *vi* made it tough to enter control characters, like Escape and Return — characters you use constantly in keymaps. You had to type Control-V before each control character that you wanted to store in the macro. Also, *vi* represents stored control characters as a two-character sequence: `^[` for Escape, for instance, and `^M` for Return. But, if you actually typed `^[` or `^M` on the keyboard, *vi* wouldn't recognize those as Escape or Return; instead, it would treat the `^` as a "go to start of line" command and the `[` as a, well, whatever a left bracket does. So, you generally can't copy a keymap with your mouse, then paste it in later.

vim accepts the old Control-V syntax but also understands new representations. Among others, `<Esc>` in a keymap stands for the Escape key, `<F3>` represents the F3 key, and `<CR>` stands for RETURN. As an example, here's a keymap for the F3 key that adds three lines below the current line: a blank line, the line `^^^ NOTE ^^^`, and another blank line:

```
:map <F3> o<CR>^^^ NOTE ^^^<CR><Esc>
```

As in *vi*, you can list currently-defined keymaps with the command `:map`. How do you tell the difference between literal characters and character representations in a keymap? *Vim* uses blue for character representations.

Figure One shows an example output from `:map`: an old *vi* keymap named `K` that breaks the current line before column 80, and the new `F3` keymap that was just defined. Note that the `<F3>`, `<CR>`, and `<Esc>` are all in blue, which means each represents a single character.

Keymap Modes

In *vim*, the *:map* command actually defines the keymap in three modes; *Normal*, *Visual*, and *Operator-pending*. Normal mode is the same as in *vi*: keys you type invoke commands. Visual mode is like Normal mode, but movement commands extend a highlighted (selected) area. Operator-Pending mode comes when you've typed an operator such as `d` and *vim* is waiting for you to enter a motion command. (For more info, type `:help vim-modes`.)

So, for instance, the following keymap maps the Space key to move the cursor to the next whitespace character, either a Space or Tab. It does this by searching with the `/` command and a regular expression:

```
:map <Space> /[<Space><Tab>]<CR>
```

```
K               0801Bhr<CR>
<F3>            o<CR>^^^ NOTE ^^^<CR><Esc>
```

FIGURE ONE: Displaying *vim* keymaps

(You don't need to type the character representations for Space and Tab inside the brackets. Simply press the Space and Tab keys. But you do need to type `<Space>` for the keymap name, and similarly when you remove the keymap by typing `:unmap<Space>`. Typing a literal Space for the keymap name, instead of `<Space>`, causes an error.

If you're used to using the Space key to move through a line of text instead of using the *vi* command `l` (lowercase "L"), this keymap will cause you problems. In that case, you might not want the keymap to work in Normal mode, but only in Operator-pending mode. That way, you can type `d` followed by a Space character to delete up to the next whitespace, or `10d` and Space to delete to the tenth occurrence of whitespace, leaving Space by itself to move to the next character on the current line. You'd want to use the `:omap` (Operator-pending map) command instead of `:map`.

First, remove the Space keymap by typing `:unmap <Space>`. Now type (or copy and paste from this article's browser window):

```
:omap <Space> /[<Space><Tab>]<CR>
```

Then try, say, `5d` followed by Space. However, typing SPACE by itself should move your cursor along the current line because the *omap* won't take effect here.

For more about this, type `:help 40.1` to go to section 40.1 of *vim* 's built-in help.

Introducing vim Scripting

Keymaps are great for simple programming, but they can't make more than the simplest decisions. (For instance, a keymap aborts if any part of it fails, such as a text search not finding a match.) Keymaps can do recursion — repeatedly invoking themselves and/or other keymaps — but there's little control. Other limitations apply, too.

When original *vi* users need a more-complex edit, they can filter their buffer through a *Linux* utility like *sort*, *cut*, or *perl*. For instance, if you're editing a file where the first column in each line has a number and the rest of the line is text, the following *vi* command uses *awk* to total the numbers in the first column and write the total under the last line:

```
:$r !awk '{sum += $1} END {print sum}' %
```

Or you can add a number before each line with:

```
:% !cat -n
```

Writing an *awk* script on the editor's command line can be tedious, slow, and error-prone, though. *vim* makes the job easier because you can edit its command line (for instance, use the left and right arrow keys while you type a command), and it also has command history (try the up and down arrow keys after typing a few commands). But *vim* also has a built-in scripting language that can handle many of the jobs that required external utilities in original *vi*.

You can write a *vim* script on its command line, or store a script in a file and read it with the *:source* command, as in `:source ~/myvimscript`. Third, you can store *vim* scripts in the *.vimrc* file in your home directory— a file that *vim* reads each time it starts.

Here's a simple *.vimrc* file:

```
set syntax=on
echo "You're in the" getcwd() "directory."
```

The first line sets syntax highlighting. (To find out more, type `:help 'syntax'`.) The second is a simple *vim* script command that outputs a message as you start *vim*. The two quoted strings are output literally, and `getcwd()` is a function call that returns the name of your current directory.

Scripts and Functions

Let's say that you have a file that describes the steps required to complex a task or procedure. You don't want to number the steps until the procedure has been finished. Some of the steps are on a single line, and other steps fill several lines. Let's write a keymap for `\n` that inserts a sequential number at the start of a line. Each time you type `\n`, *vim* will insert the next-higher number at the front of the line.

vim has a built-in scripting language that can handle many of the jobs that required external utilities in original *vi*

vim scripting supports both string and numeric values in a variable. A string is surrounded by double quotes (`"`) and a number isn't; an unquoted alphanumeric is a variable name. (For more information, type `:help variables`.) To set a variable, use the `:let` command. The syntax is: `:let` *var=value*.

Let's use a variable named `stepnum` to hold the step number. Set the initial value from the command line by typing a colon, `:`, and then `let stepnum=0`:

```
:let stepnum = 0
```

Next, let's write a user-defined function that increments *stepnum* by 1. *vim*' s function syntax is:

```
:function Name(var1, var2, ...)
:    body
…
:endfunction
```

The function name must start with a capital letter. The optional `return` statement returns a value from the function.

Here's the function definition.

```
:function Nextstep()
:   let g:stepnum = g:stepnum + 1
:   return g:stepnum
:   endfunction
```

You can type it during an interactive *vim* session, if you'd like; *vim* automatically prompts with another colon and two spaces until you type `endfunction`.

A function or a script can loop, test conditions and branch, open GUI confirmation boxes, test files, let you test and use vim's internal buffers, and more

All variables are local to a function unless you prefix them with `g:`, as was done with `stepnum`. `Nextstep()` returns the new value of *stepnum*.

Now for some pondering: If you simply run `Nextstep()` from the command line, it increments `g:stepnum` and does nothing else. Somehow *Nextstep()* must return a value as part of an expression that becomes an editor command.

There's another challenge: you run a function from the command line (in Command-line mode) but want the function's result to be used in Normal mode— as part of a command that inserts text in Insert mode.

You need two *vim* commands:

- `normal` executes Normal-mode commands that you type on the command line (after a colon prompt). Its argument is the normal-mode commands you want to execute. If you don't terminate the normal-mode commands, *normal* will add an `<Esc>` or `<C-C>` (CTRL-C) for you.

For example, if you wanted to insert the text `NOTE:` at the start of the current line, you'd use *vim*'s `I` (insert at start of line) command, followed by the text, followed by Escape. You can do that from a command line, with `:normal`, like this:

```
:normal "INOTE:"
```

- *execute* evaluates its argument and runs the result as an command. The argument is a string, a *vim* expression, or a combination. For example, if the variable *count* contains a number, you could move the cursor ahead by *count* words with the following command:

```
:execute "normal " count . "w"
```

vim's `.` (dot) operator does concatenation. So, if *count* contains the number 8, this would execute the command `8w` to move the cursor forward eight words. If this is moving a bit too quickly, try `:help normal` and `:help execute`, then experiment a bit.

Back to the step-numbering problem. So far, the variable *stepnum* is 0 and the function `Nextstep()` has been defined. Using `:execute`, you can run the following:

```
:execute "normal I" . Nextstep() . ". "
```

If *stepnum* is set to 3, the *normal* command emitted would be:

```
I3. <Esc>
```

That's `I` concatenated with the return value of `Nextstep()`, concatenated with a dot and a space. As before, *normal* adds the Escape automatically.

Next, define a keymap named `\n` to run that `:execute` command. Here's the keymap:

```
:map \n :execute "normal I" . Nextstep() . ". "
```

Try it! Typing `\n` from Normal mode should insert a step number on the current line. Move down to the next step and type `\n` to insert the next number.

Much more…

There's much more to *vim* functions and script writing. *vim* comes with many built-in functions.

A function or a script can loop, test conditions and branch, open GUI confirmation boxes, tell you what text is under the cursor, test files, let you test and use *vim*'s internal buffers, and more.

Chapter 41 of the *vim* online help (as of this writing) covers scripts and functions. Typing `:help usr_41` should take you there directly. Or, even better, start with the table of contents by typing `:help` and page down a few screens to read more about the powerful new features that have made *vim VI IMproved*.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for 25 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.