Christos Georghiou, Fotolia

Problem solving with Vim

TEST QUESTIONS

The Vim editor supports Perl plugins that let users manipulate the text they have just edited. Complex functions can be developed far faster than with Vim's integrated scripting language. **BY MICHAEL SCHILLI**

If somebody applies for an entry-level job in Yahoo's Perl division and has the pleasure of visiting me for a job interview, I might just ask the candidate the following question: "How do you add line numbers to a Perl listing, such as the ones typically used by computer magazines?"

Stress

Adding line numbers to a Perl listing is a fairly simple task, and almost any candidate will solve it. But some applicants trip up over the task of aligning the line numbers. If the listing has nine lines, all of the line numbers are single digit, but numbers are double digit for listings of between 10 and 99 lines, with the single-digit numbers padded with leading zeros (01-09). Longer listings with more than 100 and fewer than 1000 lines need three-digit line numbers, and programmers start numbering at 001.

Perl's integrated *printf()* function has an option to format numbers with leading zeros. A format string like *%03d* converts the integer *3* to the string *003*; *99* is converted to *099*, and *100* stays *100*.

But how does *printf()* pad the string to a variable length? If somebody suggests an *if/elsif* construction that checks for a limited number of number lengths, the alarms ring and the trap door to the shark pool opens.

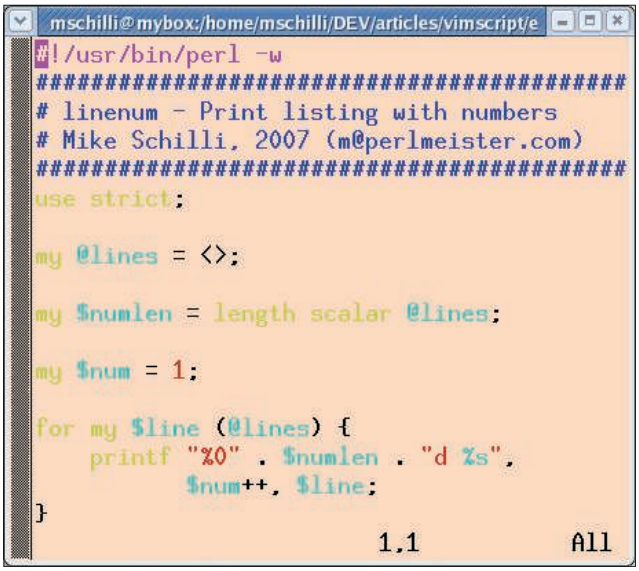

Figure 1: The listing without...

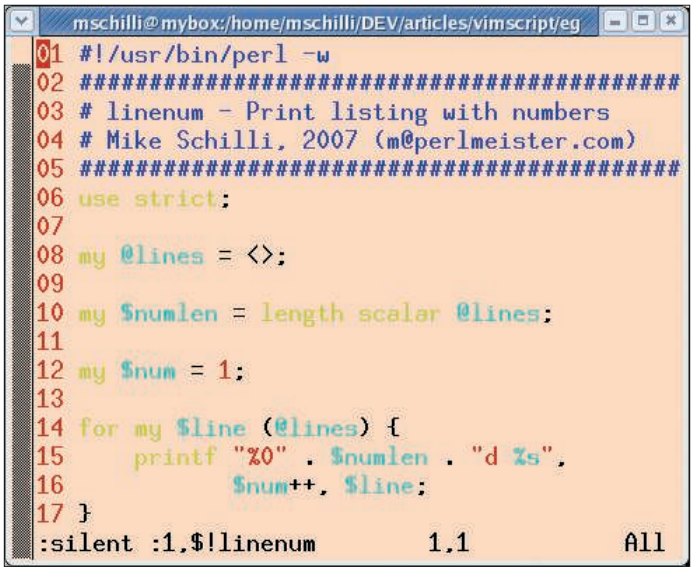

Figure 2: ...and with aligned line numbers, at the press of a button.

If the column width of the greatest line number is stored in the *$numlen* variable, you might just piece together the format string before passing it to *printf*. Concatenating the bits *"%0" . $numlen . "d"* will do the trick.

Note that if you wrote everything in a single string without paying special attention, Perl would interpret *"%0$numlend"* as referencing a non-existing variable named *$numlend*. Perl savants will know that a scalar can be written as *${numlen}* instead of *$numlen*, and this saves the day – *"%0${numlen}d"* correctly references *$numlen* and appends *"d"* at the end.

Those of you who grew up on C might remember that *printf()* supports variable format fields with the asterisk (*.**) as a wildcard and an additional parameter. An expression like *printf("%0.*d", 3, 1)* pads the number *1* to three digits by adding leading zeros.

Also, programmers can replace *3* with a variable to pad a line number to a dynamically assigned width.

Back to School

How do you know the length *$numlen* of the last line number *$num*? This is, how many digits make up *$num*? As you might know, Perl has no problem converting numbers to strings, and the integrated *length()* function will give you the string length. A call to *length($num)* therefore returns the value needed for *$numlen*.

As another option, in the decimal system, digits are weighted from right to left: 10^0, 10^1, 10^2, and so on. Thus, the number 15 can be broken down to 5 * 10^0 + 1 * 10^1. The number 100, a three-digit number, can also be expressed as 1 * 10^2. The number 1000, a four-digit numbers, is equivalent to 1 * 10^3.

So how many digits does the number *N* comprise?

If you paid attention in school, you'll probably recall that if you want to calculate the result of "10 to the power of what is *N*?", you need to establish the decimal logarithm of *N*. Although Perl does not have a decimal logarithm, it does have the *log()* function that calculates the log of a number to base *e* (the Euler number). And if you have a memory like an elephant, you might also recall that the logarithm of *N* to base *x* – that is, *log_x N* – can be calculated by dividing *log_y N* by *log_y x*.

In the example case, I can calculate the decimal logarithm of *N* in Perl as *log(N)/log(10)*. The length of the number *N* is the result of the logarithmic operation rounded down to the nearest integer and then incremented by 1. Note, though, that because of the sloppy way that your computer calculates logarithms, it might create a small inaccuracy; if the return is something like 10.999999999 instead of 11, you're in for an off-by-one error.

Listing 1: linenum

```
01 #!/usr/bin/perl -w
02 use strict;
03
04 my @lines = <>;
05
06 my $numlen =
07   length scalar @lines;
08
09 my $num = 1;
10
11 for my $line (@lines) {
12   printf "%0" . $numlen
13     . "d %s", $num++, $line;
14 }
```

Linenum Script

The *linenum* script(Listing 1) shows one approach to the problem. This approach starts by loading the lines of a script read from a file or from STDIN into the *@lines* array. Of course, this only makes sense for smaller files, but if you write Perl scripts with more than 100,000 lines, you might want to reconsider your career options anyway. Perl's dot operator (".") builds the format string, giving something like "%02d %s".

The neat thing about this interview question is that it is not just a useless puzzle that a job candidate might have heard already or can solve randomly despite the stress of the exam situation. If the candidate does not immediately see the way out, the examiner can help and at the same time discover how the candidate reacts to suggestions.

Many solutions are possible, each with its own benefits and drawbacks. What happens if you have a 10GB file? Which

approach will be fastest? What extra steps are needed to handle a file with Unicode characters?

External Perl

How can you do this in Vim – to press a button and number a complete listing in one fell swoop? The easiest solution is to run the lines in a range through the script as a filter.

A command like *:1,$!linenum* picks up all the lines in the edited file (from line 1 to the last line $) and uses STDIN to send them to the *linenum* script, which is executed as an external program thanks to the exclamation point. Vim picks up the output from the script and replaces the original lines with the results of the filter. The following *map* command in *.vimrc* maps the command to the *L* key in normal mode,

```
:map L :silent ➥
:1,$!linenum<Return>
```

assuming that *linenum* is executable and in the path for the current shell.

Listing 2: vimperl

```
01 :function! Linenum()
02
03  :if !has('perl')
04   :echo "Sorry, no Perl!"
05   return
06  :endif
07
08  perl <<EOT
09    $numlen = length(
10            $curbuf->Count());
11    for $num
12       (1..$curbuf->Count()) {
13      $newline =
14          sprintf "%0.*d %s",
15            $numlen, $num,
16            $curbuf->Get($num);
17    $curbuf->Set($num,
18                 $newline);
19    }
20  EOT
21  :endfunction
22
23 :command! Linenum :call
   Linenum()
```

The *:silent* option suppresses screen output, helping the command to run smoothly without Vim outputting status messages on the console and prompting the user to confirm. The *<Return>* command simulates a Return/Enter key press. If you left this out, Vim would fill the command line and wait for you to press Enter to confirm.

If you only want to add line numbers to a section of the document between marker *a* and marker *b* instead of the whole document, you would define the area as *'a,'b*, rather than *1,$*.

Perl Interpreter

Vim also has a built-in Perl interpreter, but you must configure the feature explicitly before compiling Vim. Vim knows at run time whether the Perl interpreter is available and provides this information via the function *has('perl')*, which returns True if Perl exists.

Running this check in your Vim scripts before use of a Perl function makes sure that Vim stops with a clearly readable error message if Perl is missing, instead of tripping over commands it is unable to understand.

Vimperl Script

The *vimperl* script (Listing 2) uses the Vim scripting language to define a Vim function called *Linenum()*, which inserts the aligned line numbers into the file you are editing. Note that Vim insists that user-defined functions start with an uppercase letter. If *has('perl')* tells you that the Perl interpreter is not installed, Vim's *:echo* command outputs an apology at the status line.

Vim's Perl documentation [3] says that *$curbuf* in the Vim Perl interpreter automatically points to the current edit buffer, which contains the lines of the file you are currently editing. The *Count()* method returns the number of lines in the buffer as an integer.

The *for* loop that follows iterates through all of the lines in the current buffer, using *$curbuf->Get($num)* to read each line, where *$num* represents the number of the buffer line currently being processed.

The *$curbuf->Set()* function then sends the line, with its shiny new number, back to the buffer. The function expects the line number and the new content as arguments.

In Vim's scripting language, comments start with double quotes and apply to the end of the current line.

Calling *:source vimperl* in Vim loads the *vimperl* script, but you should either add it to Vim's initialization file, *.vimrc*, or store it in one of the Vim plugin directories for production use.

Function!

During testing, it is practical to use *:function!* (including the exclamation point) to define Vim functions, telling Vim to overwrite the function without failing if it is already defined. Otherwise, the load process is canceled and an error message is displayed.

Within the Vim script, the Perl script is available in a here document that starts with *<<EOT* and ends with *EOT*. Note that the final *EOT* must be at the start of a line for Perl to recognize the end of the script.

After *:endfunction* indicates the end of the Vim function definition, *vimperl* calls *:command* to define a *Linenum* command, which users can call by entering *:Linenum* at the Vim command line or even map to a key. Vim again insists on the first letter of the user-defined command being uppercase. The *:map L :Linenum <Return>* command maps the command to the *L* key in normal mode.

Okay, you know my job interview tricks now, so I'll have to think of another question. I'll be looking for something just as difficult, but even more entertaining! ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/86*

[2] Vim project homepage: *http://www.vim.org*

[3] Vim's spartan Perl documentation: *http://www.vim.org/htmldoc/if_perl.html*

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.