**https://developer.ibm.com/technologies/linux/tutorials/l-vim-script-3/**

**Example 2: Enhancing Vim’s completion facilities**

For example, if you had the preceding two paragraphs in a buffer, and then—in insertion m you typed:



My use of Vim is increasingly so<CTRL-N>

Vim would search the text and determine that the only word beginning*"so..."***w**asith *sophisticated*, and would complete that word immediately:



My use of Vim is increasingly sophisticated\_

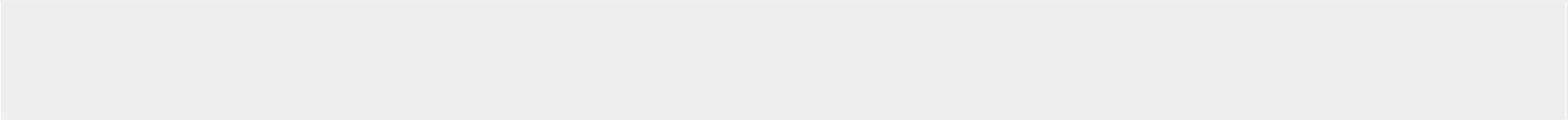
On the other hand, if you typed:



My repertoire of editing skills is bu<CTRL-N>

Vim would detect three possible completions:*built*,*buffer*, and*buffers*. By default, it would show a menu of alternatives:

**Listing 21. Text completion with alternatives**



My repertoire of editing skills is bu\_ built buffer buffers

and you could then use a sequence**CTRL**of**-N** and **CTRL-P** (or the up- and down-arrows) to step through the menu and select the word you wanted.

To cancel a completion at any time, you can**CTRL**type**-E** ; to accept and insert the currently selected alternative, you can**CTRL**type**-Y** . Typing anything else (typically, a space or newline also accepts and inserts the currently selected word, as well as whatever extra character typed.

**Designing smarter completions**

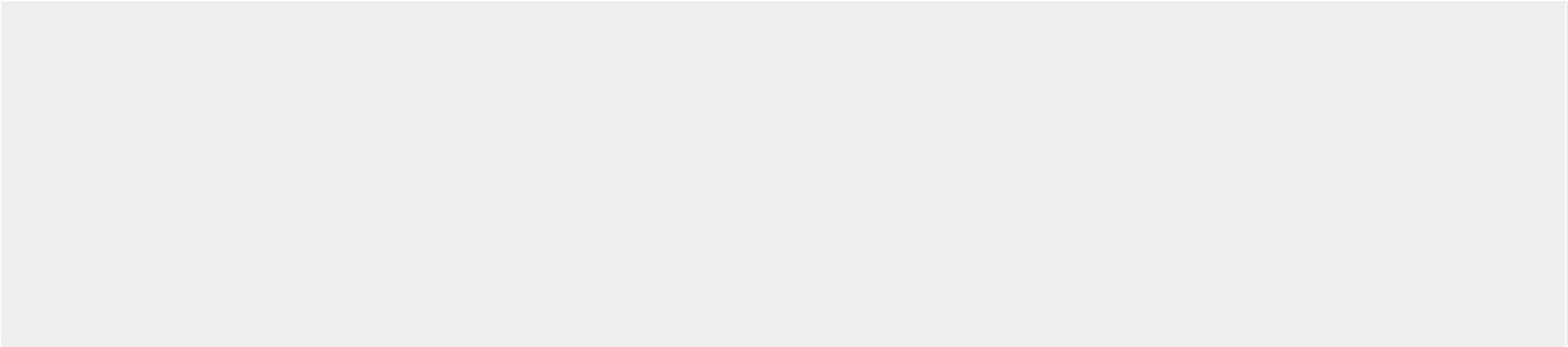
By default, it matches only sequences of "keyword" characters (alphanumerics and underscore), and it has no deep sense of context beyond matching what’s immediately to th of the cursor.

Happily, with Vimscript, we can easily remedy those deficiencies. Let’s**TAB**redefinekeyin the insertion mode so that it can be taught to recognize patterns in the text on either side and select an appropriate completion for that context. We’ll also arrange it so that, if mechanism doesn’t recognize the current insertion context, it will fall back**CTRL**to**-**Vim’s bui **N** completion mechanism. Oh, and while we’re at it, we should probably make sure we can sthe **TAB** key to type tab characters, where that’s appropriate.

**Specifying smarter completions**

To build this smarter completion mechanism, we’ll need to store a series of "contextual r to a completion request. So we’ll need a list. Or rather, a list of lists, given each con response will itself consist of four elements. Listing 22 shows how to set up that data s

**Listing 22. Setting up a look-up table in Vimscript**



* Table of completion specifications (a list of lists)...

let s:completions = []

* Function to add user-defined completions...

function! AddCompletion (left, right, completion, restore)

call insert(s:completions, [a:left, a:right, a:completion, a:restore])

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| endfunction |  |  |  |  |  |
| let s:NONE = "" |  |  |  |  |  |
| " Table of completions... | |  |  |  |  |
| " | Left | Right | Complete with... | Restore |  |
| " | ===== | ======= | ==================== | ======= |  |
| call AddCompletion( | '{', | s:NONE, | "}", | 1 | ) |
| call AddCompletion( | '{', | '}', | "\<CR>\<C-D>\<ESC>O", | 0 | ) |

Scripting the Vim editor, Part 3: Built-in lists Page 11 of19

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| developerWorks® |  |  |  |  | ibm.com/developerWorks/ | |
|  |  |  |  |  |  |  |
| call AddCompletion( | '\[', | s:NONE, | "]", | 1 | ) |  |
| call AddCompletion( | '\[', | '\]', | "\<CR>\<ESC>O\<TAB>", | 0 | ) |  |
| call AddCompletion( | '(', | s:NONE, | ")", | 1 | ) |  |
| call AddCompletion( | '(', | ')', | "\<CR>\<ESC>O\<TAB>", | 0 | ) |  |
| call AddCompletion( | '<', | s:NONE, | ">", | 1 | ) |  |
| call AddCompletion( | '<', | '>', | "\<CR>\<ESC>O\<TAB>", | 0 | ) |  |
| call AddCompletion( | '"', | s:NONE, | '"', | 1 | ) |  |
| call AddCompletion( | '"', | '"', | "\\n", | 1 | ) |  |
| call AddCompletion( | "'", | s:NONE, | "'", | 1 | ) |  |
| call AddCompletion( | "'", | "'", | s:NONE, | 0 | ) |  |

The list-of-lists we create will act as a table of contextual response specifications, an stored in the list s:completions. Each entry in the list will itself be a list, with fo

* A string specifying a regular expression to match what’s to the left of the cursor
* A string specifying a regular expression to match what’s to the right of the cursor
* A string to be inserted when both contexts are detected
* A flag indicating whether to automatically restore the cursor to its pre-completion after the completion text has been inserted

To populate the table, we create a small AddCompletion(). This function expects four arguments: the left and right contexts, and the replacement text,restore cursor" flag. The series of arguments are simply collected into a single list:



[a:left, a:right, a:completion, a:restore]

and that list is then prepended as a single element at thes:completionsthevariable using the built insert() function:



call insert(s:completions, [a:left, a:right, a:completion, a:restore])

Repeated calls AddCompletion() therefore build up a list of lists, each of which specifie completion. The code in Listing 22 does the work.

The first callAddCompletion():

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| " | Left | Right | Complete with... | Restore |  |
| " | ===== | ======= | ==================== | ======= |  |
| call AddCompletion( '{', | | s:NONE, | '}', | 1 | ) |

specifies that, when the new mechanism encounters a curly brace to the left of the cursor nothing to the right, it should insert a closing curly brace and then restore the cursor completion position. That is, when completing:



while (1) {\_

(where the \_ represents the cursor), the mechanism will now produce:



while (1) {\_}

leaving the cursor conveniently in the middle of the newly closed block.

Scripting the Vim editor, Part 3: Built-in lists Page 12 of19

The second call AddCompletion():

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| " | Left | Right | Complete with... | Restore |  |
| " | ===== | ======= | ==================== | ======= |  |
| call AddCompletion( | '{', | '}', | "\<CR>\<C-D>\<ESC>O", | 0 | ) |

then proceeds to make the completion mechanism smarter still. It specifies that, when the mechanism encounters an opening curly brace to the left of the cursor and a closing brace right of the cursor, it should insert a newline, outdent the closing**CTRL-D**curly),then(viaescape from insertion mode**ESC**( ) and open a new line above the closing Ocurly). (

Assuming the smartindent" option is enabled, the net effect of the sequence is that, when press **TAB** in the following context



while (1) {\_}

the mechanism will produce:



while (1) {

\_

}

In other words, because of the first two additions to the completion**TAB**-completiontable, after an opening brace closes it on the same line, and then immediately doing**TAB**a -secondcompletion "stretches" the block across several lines (with correct indenting).

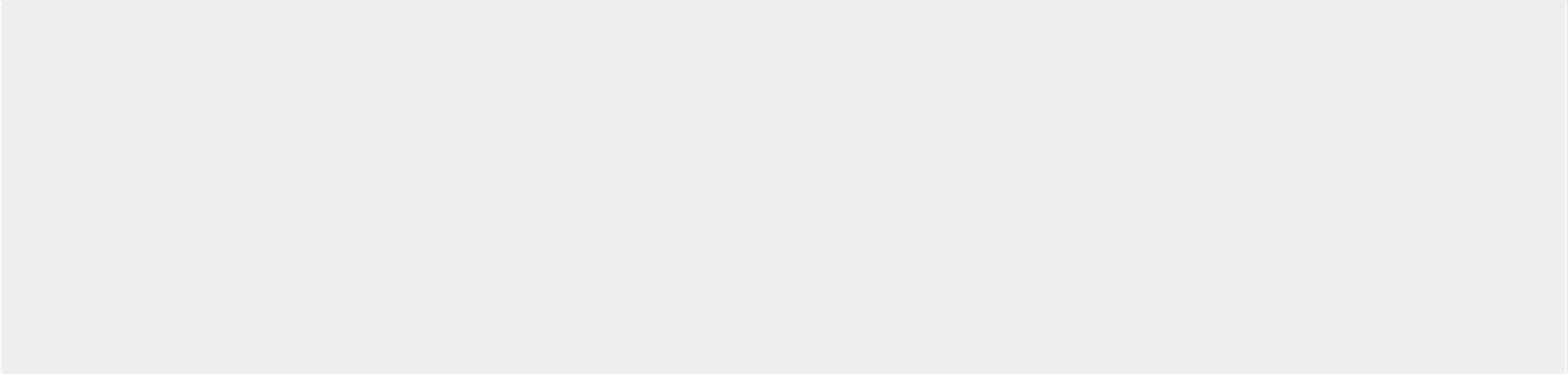
The remaining callsAddCompletion() replicate this arrangement for the three other kinds o brackets (square, round, and angle) and also provide special completion semantics for sin and double-quotes. Completing after a double-quote appends the matching double-quote, whi completing between two double quotes appends\n a(newline) metacharacter. Completing after

a single quote appends the matching single quote, and then a second completion attempt do nothing.

**Implementing smarter completions**

Once the list of completion-specifications has been set up, all that remains is to implem function to select the appropriate completion from the table, and then bind that**TAB**function key. Listing 23 shows that code.

**Listing 23. A smarter completion function**



" Implement smart completion magic...

function! SmartComplete ()

* Remember where we parked...

let cursorpos = getpos('.') let cursorcol = cursorpos[2] let curr\_line = getline('.')

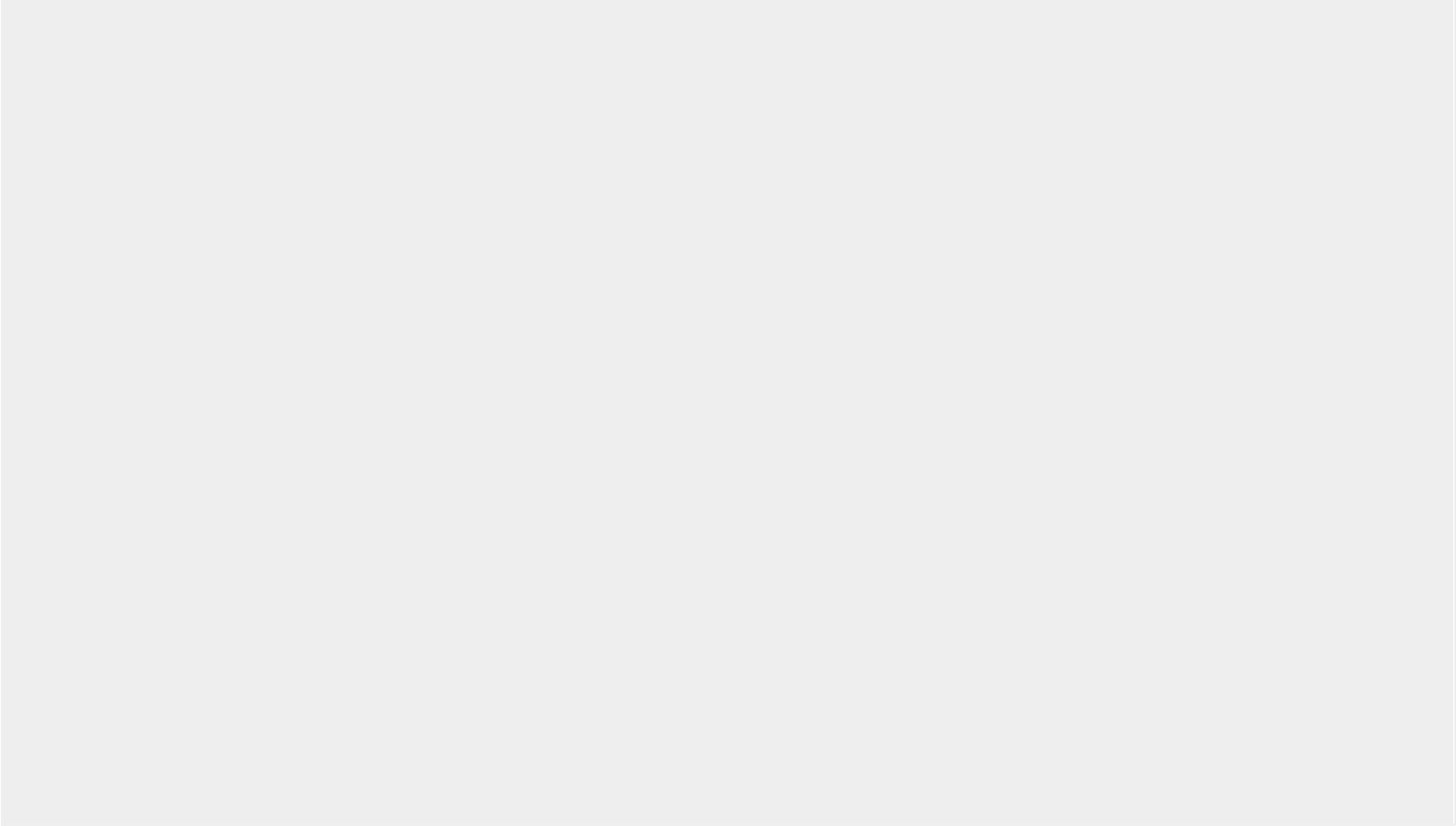
* Special subpattern to match only at cursor position...

let curr\_pos\_pat = '\%' . cursorcol . 'c'

* Tab as usual at the left margin...

if curr\_line =~ '^\s\*' . curr\_pos\_pat return "\<TAB>"

endif



" How to restore the cursor position...

let cursor\_back = "\<C-O>:call setpos('.'," . string(cursorpos) . ")\<CR>"

" If a matching smart completion has been specified, use that...

for [left, right, completion, restore] in s:completions let pattern = left . curr\_pos\_pat . right

if curr\_line =~ pattern

" Code around bug in setpos() when used at EOL...

if cursorcol == strlen(curr\_line)+1 && strlen(completion)==1 let cursor\_back = "\<LEFT>"

endif

" Return the completion...

return completion . (restore ? cursor\_back : "") endif

endfor

" If no contextual match and after an identifier, do keyword completion...

if curr\_line =~ '\k' . curr\_pos\_pat return "\<C-N>"

" Otherwise, just be a <TAB>...

else

return "\<TAB>" endif

endfunction

" Remap <TAB> for smart completion on various characters...

inoremap <silent> <TAB> <C-R>=SmartComplete()<CR>

The SmartComplete() function first locates the cursor, using getpos()builtfunction-in with a

'.' argument (that is, "get position of cursor"). That call returns a list of four elemen number (usually zero), the row and column numbers (both indexed from 1), and a special "v offset" (which is also usually zero, and not relevant here). We’re primarily interested i two values, as they indicate the location of the cursor. SmartComplete() needs the

column number, which is extracted by indexing into thegetpos()thatreturned, like so:



let cursorcol = cursorpos[2]

The function also needs to know the text on the current line, which can be retrieved usin getline(), and is storedcurr\_line.

SmartComplete() is going to convert each entrys:completions table into a pattern to be matched against the current line. In order to correctly match left and right contexts aro cursor, it needs to ensure the pattern matches only at the cursor’s column. Vim has a spe subpattern for that:\%Nc (whereN is the column number required). So, the function creates th subpattern by interpolating the cursor’s column position found earlier:



let curr\_pos\_pat = '\%' . cursorcol . 'c'

Because we’re eventually going to bind this function**TAB** tokey,thewe’d like the function to stil insert **TAB**a whenever possible, and especially at the start ofSmartComplete() first

checks if there is only whitespace to the left of the cursor position, in which case it r tabspace:

if curr\_line =~ '^\s\*' . curr\_pos\_pat return "\<TAB>"

endif

If the cursor isn’t at the start of SmartComplete() needs to check all the entries in the completion table and determine which, if any, apply. Some of those entries will specify t cursor should be returned to its previous position after completion, which will require e a custom command from within insertion mode. That command is simply a call to the built-i setpos() function, passing the value the original information from thegetpos(). call to

To execute that function call from within insertion mode**CTRL**requires**-O**escape (see:help i\_CTRL-O in any Vim session).SmartComplete() prebuilds the necessary**CTRL-O** command as a string and storescursor\_back:



let cursor\_back = "\<C-O>:call setpos('.'," . string(cursorpos) . ")\<CR>"

**A more-sophisticated for loop**

To walk through the completions table, the function uses a special versionforstatementofthe.

The standardfor loop in Vimscript walks through a one-dimensional list, one element at a

**Listing 24. A standard for loop**

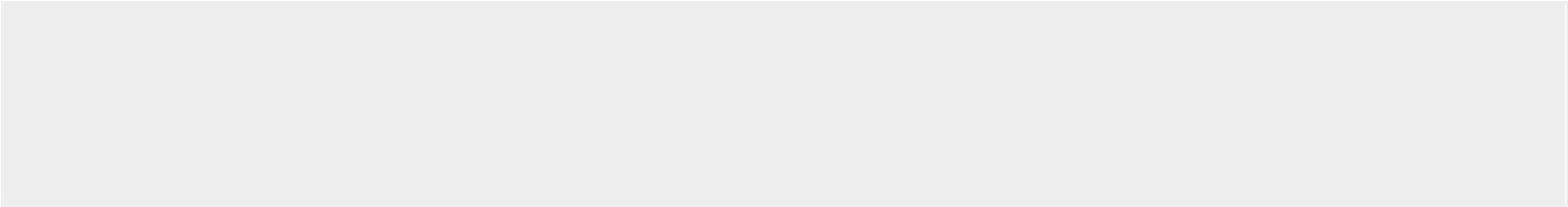


for name in list echo name

endfor

However, if the list is two-dimensional (that is, each element is itself a list), then yo "unpack" the contents of each nested list as it is iterated. You could do that like so:

**Listing 25. Iterating over nested lists**



for nested\_list in list\_of\_lists

|  |  |  |  |
| --- | --- | --- | --- |
| let name | | = nested\_list[0] | |
| let | rank | = | nested\_list[1] |
| let | serial = | | nested\_list[2] |

echo rank . ' ' . name . '(' . serial . ')' endfor

but Vimscript has a much cleaner shorthand for it:

**Listing 26. A cleaner shorthand for iterating over nested lists**



for [name, rank, serial] in list\_of\_lists

echo rank . ' ' . name . '(' . serial . ')' endfor

On each iteration, the loop takes the next nestedlist\_of\_lists and assigns the first element of that nested listnameto, the second nested elementrank , and the thirdserial.

Using this special formforofloop makes it easy SmartComplete() to walk through the table of completions and give a logical name to each component of each completion:



for [left, right, completion, restore] in s:completions

**Recognizing a completion context**

Within the loop,SmartComplete() constructs a regular expression by placing the left and righ context patterns around the special subpattern that matches the cursor position:



let pattern = left . curr\_pos\_pat . right

If the current line matches the resulting regex, then the function has found the correct (the text of which is already in completion) and can return it immediately. Of course, it to append the cursor restoration command it built earlier, if the selected completion has it (that restore is true).

Unfortunately, setpos() -based cursor restoration command has a problem. In Vim versions 7.2 or earlier, there’s an obscure idiosyncrasysetpos()in: it doesn’t correctly reposition the cursor in insertion mode if the cursor was previously at the end of a line and the comple to be inserted is only one character long. In that special case, the restoration command changed to a single left-arrow, which moves the cursor back over the one newly inserted c

So, before the selected completion is returned, the following code makes that change:

**Listing 27. Restoring the cursor after a one-character insertion at end-of**



if cursorcol == strlen(curr\_line)+1 && strlen(completion)==1 let cursor\_back = "\<LEFT>"

endif

|  |  |  |
| --- | --- | --- |
| All that remains is to | return the selected completion, | cursor\_backthe command if |
| cursor restoration was | requested: | |



return completion . (restore ? cursor\_back : "")

If none of the entries from the completion table match the currentSmartComplete()

will eventually fall outfor loopthe and will then try two final alternatives. If the charact

immediately before the cursor was a "keyword" character, it invokes a normal keyword-comp by returning **CTRL**a**-N** :

**Listing 28. Falling back to CTRL-N behavior**

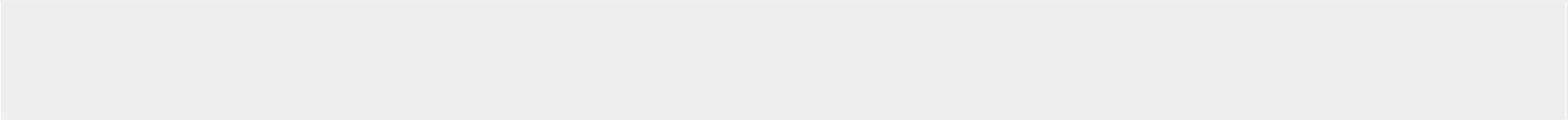


" If no contextual match and after an identifier, do keyword completion...

if curr\_line =~ '\k' . curr\_pos\_pat return "\<C-N>"

Otherwise, no completion was possible, so it falls back to acting **TAB**likekey,anormalby returning a literal tab character:

**Listing 29. Falling back to normal TAB key behavior**



" Otherwise, just be a <TAB>...

else

return "\<TAB>" endif

**Deploying the new mechanism**

Now we just have to make the**TAB** key callSmartComplete() in order to work out what it should insert. That’s done withinoremap, like so:



inoremap <silent> <TAB> <C-R>=SmartComplete()<CR>

The key-mapping converts any insert-mode**TAB** to a**CTRL-R=** , callingSmartComplete() and inserting the completion string it returns:help i\_CTRL-R or the [first article in](http://www.ibm.com/developerworks/linux/library/l-vim-script-1/index.html) thisforseries details of this mechanism).

The inoremap form ofimap is used here because some of the completion strings that SmartComplete() returns also contain**TAB**a character. If a regularimapwere used, inserting

that returned**TAB** would immediately cause this same key-mapping to be re-invoked, calling SmartComplete() again, which might return another**TAB**, and so on.

With theinoremap in place, we now have**TAB**a key that can:

* Recognize special user-defined insertion contexts and complete them appropriately
* Fall back to regular**CTRL-N** completion after an identifier
* Still act like**TAB** a everywhere else

In addition, with the code from Listings 22 and 23 placed in your .vimrc file, you will b add new contextual completions simply by extending the completion table with extra calls AddCompletion(). For example, you could make it easier to start new Vimscript functions w



call AddCompletion( 'function!\?', "", "\<CR>endfunction", 1 )

|  |  |  |  |
| --- | --- | --- | --- |
| so that | tabbing immediately after a function keyword appends the |  |  |
| endfunction |  |
| keyword | on the next line. |  |  |

Or, you could autocomplete C/C++ comments intelligently (assumingcindenttheoption is also set) with:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| call | AddCompletion( | '/\\*', | "", | '\*/', | 1 ) |
| call | AddCompletion( | '/\\*', | '\\*/', | "\<CR>\* \<CR>\<ESC>\<UP>A", | 0 ) |

So that:



/\*\_<TAB>

appends a closing comment delimiter after the cursor:



/\*\_\*/

and a second**TAB** at that point inserts an elegant multiline comment and positions the cur the middle of it:

/\*

\* \_

\*/