What is your most productive shortcut with Vim?

I've heard a lot about Vim, both pros and cons. It really seems you should be (as a developer) faster with Vim than with any other editor. I'm using Vim to do some basic stuff and I'm at best 10 times *less productive* with Vim.

The only two things you should care about when you talk about speed (you may not care enough about them, but you should) are:

1. Using alternatively left and right hands is the **fastest** way to use the keyboard.
2. Never touching the mouse is the second way to be as fast as possible. It takes ages for you to move your hand, grab the mouse, move it, and bring it back to the keyboard (and you often have to look at the keyboard to be sure you returned your hand properly to the right place)

Here are two examples demonstrating why I'm far less productive with Vim.

Copy/Cut & paste. I do it all the time. With all the contemporary editors you press Shift with the left hand, and you move the cursor with your right hand to select text. Then Ctrl + C copies, you move the cursor and Ctrl + V pastes.

With Vim it's horrible:

- `yy` to copy one line (you almost never want the whole line!)
- `[number xx]yy` to copy `xx` lines into the buffer. But you never know exactly if you've selected what you wanted. I often have to do `[number xx]dd` then `u` to undo!

Another example? **Search & replace.**

- In PSPad: Ctrl + f then type what you want you search for, then press Enter.
- In Vim: `/` , then type what you want to search for, then if there are some special characters put `\` before *each* special character, then press Enter.

And everything with Vim is like that: it seems I don't know how to handle it the right way.

NB : **I've already read the Vim cheat sheet** :)

My question is:

What is the way you use Vim that makes you more productive than with a contemporary editor?

vim productivity vi

edited Apr 1 '10 at 5:27

community wiki
11 revs, 8 users 36%
Olivier Pons

locked by ThiefMaster ♦ Feb 17 '13 at 11:48

This question exists because it has historical significance, but **it is not considered a good, on-topic question for this site**, so please do not use it as evidence that you can ask similar questions here. This question and its answers are frozen and cannot be changed. More info: help center.

11 I think this should be re-opened as the question asks for tips on being efficient in Vim. I have never visited past the first page of the thread, but I don't see any arguing, and if new answers arise from some new Vim feature or tip in the future I did not know about before, I would like to be able to find it here. Anyhoooo, just my thoughts on it. Please consider re-opening? :) – kikuchiyo Dec 14 '12 at 22:23

|

50 Answers

1 2 next

Your problem with Vim is that you don't grok vi.

You mention cutting with `yy` and complain that you almost never want to cut whole lines. In fact programmers, editing source code, very often want to work on whole lines, ranges of lines and blocks of code. However, `yy` is only one of many way to yank text into the anonymous copy buffer (or "register" as it's called in **vi**).

The "Zen" of **vi** is that you're speaking a language. The initial `y` is a verb. The statement `yy` is a synonym for `y_`. The `y` is doubled up to make it easier to type, since it is such a common operation.

This can also be expressed as `dd P` (delete the current line and paste a copy back into place; leaving a copy in the anonymous register as a side effect). The `y` and `d` "verbs" take any movement as their "subject." Thus `yW` is "yank from here (the cursor) to the end of the current/next (big) word" and `y'a` is "yank from here to the line containing the mark named '*a*'."

If you only understand basic up, down, left, and right cursor movements then **vi** will be no more productive than a copy of "notepad" for you. (Okay, you'll still have syntax highlighting and the ability to handle files larger than a piddling ~45KB or so; but work with me here).

vi has 26 "marks" and 26 "registers." A mark is set to any cursor location using the `m` command. Each mark is designated by a single lower case letter. Thus `ma` sets the '*a*' mark to the current location, and `mz` sets the '*z*' mark. You can move to the line containing a mark using the `'` (single quote) command. Thus `'a` moves to the beginning of the line containing the '*a*' mark. You can move to the precise location of any mark using the `` ` `` (backquote) command. Thus `` `z `` will move directly to the exact location of the '*z*' mark.

Because these are "movements" they can also be used as subjects for other "statements."

So, one way to cut an arbitrary selection of text would be to drop a mark (I usually use '*a*' as my "first" mark, '*z*' as my next mark, '*b*' as another, and '*e*' as yet another (I don't recall ever having interactively used more than four marks in 15 years of using **vi**; one creates one's own conventions regarding how marks and registers are used by macros that don't disturb one's interactive context). Then we go to the other end of our desired text; we can start at either end, it doesn't matter. Then we can simply use ``d`a`` to cut or ``y`a`` to copy. Thus the whole process has a 5 keystrokes overhead (six if we started in "insert" mode and needed to Esc out command mode). Once we've cut or copied then pasting in a copy is a single keystroke: `p`.

I say that this is one way to cut or copy text. However, it is only one of many. Frequently we can more succinctly describe the range of text without moving our cursor around and dropping a mark. For example if I'm in a paragraph of text I can use `{` and `}` movements to the beginning or end of the paragraph respectively. So, to move a paragraph of text I cut it using `{ d}` (3 keystrokes). (If I happen to already be on the first or last line of the paragraph I can then simply use `d}` or `d{` respectively.

The notion of "paragraph" defaults to something which is usually intuitively reasonable. Thus it often works for code as well as prose.

Frequently we know some pattern (regular expression) that marks one end or the other of the text in which we're interested. Searching forwards or backwards are movements in **vi**. Thus they can also be used as "subjects" in our "statements." So I can use `d/foo` to cut from the current line to the next line containing the string "foo" and `y?bar` to copy from the current line to the most recent (previous) line containing "bar." If I don't want whole lines I can still use the search movements (as statements of their own), drop my mark(s) and use the `` `x `` commands as described previously.

In addition to "verbs" and "subjects" **vi** also has "objects" (in the grammatical sense of the term). So far I've only described the use of the anonymous register. However, I can use any of the 26 "named" registers by *prefixing* the "object" reference with `"` (the double quote modifier). Thus if I use `"add` I'm cutting the current line into the '*a*' register and if I use `"by/foo` then I'm yanking a copy of the text from here to the next line containing "foo" into the '*b*' register. To paste from a register I simply prefix the paste with the same modifier sequence: `"ap` pastes a copy of the '*a*' register's contents into the text after the cursor and `"bP` pastes a copy from '*b*' to before the current line.

This notion of "prefixes" also adds the analogs of grammatical "adjectives" and "adverbs' to our text manipulation "language." Most commands (verbs) and movement (verbs or objects, depending on context) can also take numeric prefixes. Thus `3J` means "join the next three lines" and `d5}` means "delete from the current line through the end of the fifth paragraph down from here."

This is all intermediate level **vi**. None of it is **Vim** specific and there are far more advanced tricks in **vi** if you're ready to learn them. If you were to master just these intermediate concepts then you'd probably find that you rarely need to write any macros because the text manipulation language is sufficiently concise and expressive to do most things easily enough using the editor's "native" language.

A sampling of more advanced tricks:

There are a number of `:` commands, most notably the `:% s/foo/bar/g` global substitution technique. (That's not advanced but other `:` commands can be). The whole `:` set of commands was historically inherited by **vi**'s previous incarnations as the **ed** (line editor) and later the **ex** (extended line editor) utilities. In fact **vi** is so named because it's the visual interface to **ex**.

`:` commands normally operate over lines of text. **ed** and **ex** were written in an era when terminal screens were uncommon and many terminals were "teletype" (TTY) devices. So it was common to work from printed copies of the text, using commands through an extremely terse interface (common connection speeds were 110 baud, or, roughly, 11 characters per second -- which is slower than a fast typist; lags were common on multi-user interactive sessions; additionally there was often some motivation to conserve paper).

So the syntax of most `:` commands includes an address or range of addresses (line number) followed by a command. Naturally one could use literal line numbers: `:127,215 s/foo/bar` to change the first occurrence of "foo" into "bar" on each line between 127 and 215. One could also use some abbreviations such as `.` or `$` for current and last lines respectively. One could also use relative prefixes `+` and `-` to refer to offsets after or before the curent line, respectively. Thus: `:.,$j` meaning "from the current line to the last line, join them all into one line". `:%` is synonymous with `:1,$` (all the lines).

The `:... g` and `:... v` commands bear some explanation as they are incredibly powerful. `:... g` is a prefix for "globally" applying a subsequent command to all lines which match a pattern (regular expression) while `:... v` applies such a command to all lines which do NOT match the given pattern ("v" from "conVerse"). As with other **ex** commands these can be prefixed by addressing/range references. Thus `:.,+21g/foo/d` means "delete any lines containing the string "foo" from the current one through the next 21 lines" while `:.,$v/bar/d` means "from here to the end of the file, delete any lines which DON'T contain the string "bar."

It's interesting that the common Unix command **grep** was actually inspired by this **ex** command (and is named after the way in which it was documented). The **ex** command `:g/re/p` (grep) was the way they documented how to "globally" "print" lines containing a "regular expression" (re). When **ed** and **ex** were used, the `:p` command was one of the first that anyone learned and often the first one used when editing any file. It was how you printed the current contents (usually just one page full at a time using `:.,+25p` or some such).

Note that `:% g/.../d` or (its reVerse/conVerse counterpart: `:% v/.../d` are the most common usage patterns. However there are couple of other `ex` commands which are worth remembering:

We can use `m` to move lines around, and `j` to join lines. For example if you have a list and you want to separate all the stuff matching (or conversely NOT matching some pattern) without deleting them, then you can use something like: `:% g/foo/m$` ... and all the "foo" lines will have been moved to the end of the file. (Note the other tip about using the end of your file as a scratch space). This will have preserved the relative order of all the "foo" lines while having extracted them from the rest of the list. (This would be equivalent to doing something like: `1G!GGmap!Ggrep foo<ENTER>1G:1,'a g/foo'/d` (copy the file to its own tail, filter the tail through `grep` , and delete all the stuff from the head).

To join lines usually I can find a pattern for all the lines which need to be joined to their predecessor (all the lines which start with "^ " rather than "^ * " in some bullet list, for example). For that case I'd use: `:% g/^   /-1j` (for every matching line, go up one line and join them). (BTW: for bullet lists trying to search for the bullet lines and join to the next doesn't work for a couple reasons ... it can join one bullet line to another, and it won't join any bullet line to *all* of its continuations; it'll only work pairwise on the matches).

Almost needless to mention you can use our old friend `s` (substitute) with the `g` and `v` (global/converse-global) commands. Usually you don't need to do so. However, consider some case where you want to perform a substitution only on lines matching some other pattern. Often you can use a complicated pattern with captures and use back references to preserve the portions of the lines that you DON'T want to change. However, it will often be easier to separate the match from the substitution: `:% g/foo/s/bar/zzz/g` -- for every line containing "foo" substitute all "bar" with "zzz." (Something like `:% s/\(.*foo.*\)bar\(.*\)/\1zzz\2/g` would only work for the cases those instances of "bar" which were PRECEDED by "foo" on the same line; it's ungainly enough already, and would have to be mangled further to catch all the cases where "bar" preceded "foo")

The point is that there are more than just `p` , `s` , and `d` lines in the `ex` command set.

The `:` addresses can also refer to marks. Thus you can use: `:'a,'bg/foo/j` to join any line containing the string foo to its subsequent line, if it lies between the lines between the '*a*' and '*b*' marks. (Yes, all of the preceding `ex` command examples can be limited to subsets of the file's lines by prefixing with these sorts of addressing expressions).

That's pretty obscure (I've only used something like that a few times in the last 15 years). However, I'll freely admit that I've often done things iteratively and interactively that could probably have been done more efficiently if I'd taken the time to think out the correct incantation.

Another very useful **vi** or **ex** command is `:r` to read in the contents of another file. Thus: `:r foo` inserts the contents of the file named "foo" at the current line.

More powerful is the `:r!` command. This reads the results of a command. It's the same as suspending the **vi** session, running a command, redirecting its output to a temporary file, resuming your **vi** session, and reading in the contents from the temp. file.

Even more powerful are the `!` (bang) and `:... !` (**ex** bang) commands. These also execute external commands and read the results into the current text. However, they also filter selections of our text through the command! This we can sort all the lines in our file using `1G!Gsort` (`G` is the **vi** "goto" command; it defaults to going to the last line of the file, but can be prefixed by a line number, such as 1, the first line). This is equivalent to the **ex** variant `:1,$!sort` . Writers often use `!` with the Unix **fmt** or **fold** utilities for reformating or "word wrapping" selections of text. A very common macro is `{!}fmt` (reformat the current paragraph). Programmers sometimes use it to run their code, or just portions of it, through **indent** or other code reformatting tools.

Using the `:r!` and `!` commands means that any external utility or filter can be treated as an extension of our editor. I have occasionally used these with scripts that pulled data from a database, or with **wget** or **lynx** commands that pulled data off a website, or **ssh** commands that pulled data from remote systems.

Another useful **ex** command is `:so` (short for `:source`). This reads the contents of a file as a series of commands. When you start **vi** it normally, implicitly, performs a `:source` on `~/.exinitrc` file (and **Vim** usually does this on `~/.vimrc` , naturally enough). The use of this is that you can change your editor profile on the fly by simply sourcing in a new set of macros, abbreviations, and editor settings. If you're sneaky you can even use this as a trick for storing sequences of **ex** editing commands to apply to files on demand.

For example I have a seven line file (36 characters) which runs a file through **wc**, and inserts a C-style comment at the top of the file containing that word count data. I can apply that "macro" to a file by using a command like: `vim +'so mymacro.ex' ./mytarget`

(The `+` command line option to **vi** and **Vim** is normally used to start the editing session at a given line number. However it's a little known fact that one can follow the `+` by any valid **ex** command/expression, such as a "source" command as I've done here; for a simple example I have scripts which invoke: `vi +'/foo/d|wq!' ~/.ssh/known_hosts` to remove an entry from my SSH known hosts file non-interactively while I'm re-imaging a set of servers).

Usually it's far easier to write such "macros" using Perl, AWK, **sed** (which is, in fact, like **grep** a utility inspired by the **ed** command).

The `@` command is probably the most obscure **vi** command. In occasionally teaching advanced systems administration courses for close to a decade I've met very few people who've ever used it. `@` executes the contents of a register as if it were a **vi** or **ex** command. Example: I often use: `:r!locate ...` to find some file on my system and read its name into my document. From there I delete any extraneous hits, leaving only the full path to the file I'm interested in. Rather than laboriously Tab-ing through each component of the path (or worse, if I happen to be stuck on a machine without Tab completion support in its copy of **vi**) I just use:

1. `0i:r` (to turn the current line into a valid **:r** command),
2. `"cdd` (to delete the line into the "c" register) and
3. `@c` execute that command.

That's only 10 keystrokes (and the expression `"cdd @c` is effectively a finger macro for me, so I can type it almost as quickly as any common six letter word).

A sobering thought

I've only scratched to surface of **vi**'s power and none of what I've described here is even part of the "improvements" for which **vim** is named! All of what I've described here should work on any old copy of **vi** from 20 or 30 years ago.

There are people who have used considerably more of **vi**'s power than I ever will.

edited Dec 22 '11 at 11:43

community wiki
18 revs, 16 users 64%
Jim Dennis

440 Holy code monkeys,..that's one in-depth answer. What's great is that you probably wrote in in vim in about 10 keystrokes. – Stimul8d Nov 27 '09 at 15:36

48 @Wahnfieden -- grok is exactly what I meant: en.wikipedia.org/wiki/Grok (It's apparently even in the OED --- the closest we anglophones have to a canonical lexicon). To "grok" an editor is to find yourself using its commands fluently ... as if they were your natural language. – Jim Dennis Feb 12 '10 at 4:08

22 wow, a very well written answer! i couldn't agree more, although i use the `@` command a lot (in combination with `q` : record macro) – knittl Feb 27 '10 at 13:15

63 Superb answer that utterly redeems a really horrible question. I am going to upvote this question, that normally I would downvote, just so that this answer becomes easier to find. (And I'm an Emacs guy! But this way I'll have somewhere to point new folks who want a good explanation of what vi power users find fun about vi. Then I'll tell them about Emacs and they can decide.) – Brandon Rhodes Mar 29 '10 at 15:26

8 Can you make a website and put this tutorial there, so it doesn't get burried here on stackoverflow. I have yet to read better introduction to vi then this. – Marko Apr 1 '10 at 14:47

You are talking about text selecting and copying, I think that you should give a look to the Vim

Visual Mode.

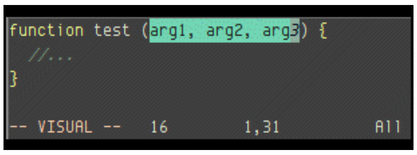
In the visual mode, you are able to select text using Vim commands, then you can do whatever you want with the selection.

Consider the following common scenarios:

You need to select to the next matching parenthesis.

You could do:

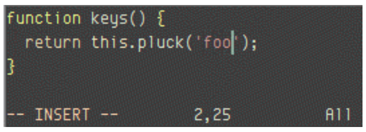
- `v%` if the cursor is on the starting/ending parenthesis
- `vib` if the cursor is inside the parenthesis block



```
function test (arg1, arg2, arg3) {
  //...
}
-- VISUAL --   16         1,31          All
```

You want to select text between quotes:

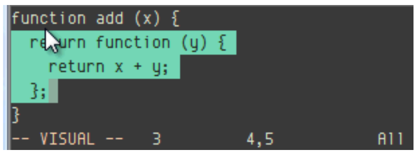
- **vi"** for double quotes
- **vi'** for single quotes



```
function keys() {
  return this.pluck('foo');
}
-- INSERT --        2,25        All
```

You want to select a curly brace block (very common on C-style languages):

- `viB`
- `vi{`

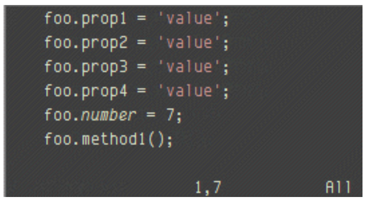


```
function add (x) {
  return function (y) {
    return x + y;
  };
}
-- VISUAL --   3         4,5           All
```

You want to select the entire file:

- `ggVG`

Visual block selection is another really useful feature, it allows you to select a rectangular area of text, you just have to press Ctrl - V to start it, and then select the text block you want and perform any type of operation such as yank, delete, paste, edit, etc. It's great to edit *column oriented* text.



```
    foo.prop1 = 'value';
    foo.prop2 = 'value';
    foo.prop3 = 'value';
    foo.prop4 = 'value';
    foo.number = 7;
    foo.method1();

                    1,7         All
```

edited Jan 27 '11 at 21:27
chelmertz
11.4k 2 28 42

answered Aug 2 '09 at 8:27
CMS
426k 118 730 746

22 Yes, but it was a specific complaint of the poster. Visual mode is Vim's best method of direct text-selection and manipulation. And since vim's buffer traversal methods are superb, I find text selection in vim fairly pleasurable. – guns Aug 2 '09 at 9:54

9 I think it is also worth mentioning Ctrl-V to select a block - ie an arbitrary rectangle of text. When you need it it's a lifesaver. – Hamish Downer Mar 16 '10 at 13:34

6 @viksit: I'm using Camtasia, but there are plenty of alternatives: codinghorror.com/blog/2006/11/screencasting-for-windows.html – CMS Apr 2 '10 at 2:07

75 Those are the coolest mini-screencasts I've seen today. Please make more! – Marius Andersen Apr 2 '10 at 10:26

3 Also, if you've got a visual selection and want to adjust it, `o` will hop to the other end. So you can move both the beginning and the end of the selection as much as you like. – Nathan Long Mar 1 '11 at 19:05

Some productivity tips:

Smart movements

- `*` and `#` search for the word under the cursor forward/backward.
- `w` to the next word
- `W` to the next space-separated word
- `b` / `e` to the begin/end of the current word. (`B` / `E` for space separated only)
- `gg` / `G` jump to the begin/end of the file.
- `%` jump to the matching { .. } or (..), etc..
- `{` / `}` jump to next paragraph.
- `'.` jump back to last edited line.
- `g;` jump back to last edited position.

Quick editing commands

- `I` insert at the begin.
- `A` append to end.
- `o` / `O` open a new line after/before the current.
- `v` / `V` / `Ctrl+V` visual mode (to select text!)
- `Shift+R` replace text
- `C` change remaining part of line.

Combining commands

Most commands accept a amount and direction, for example:

- `cW` = change till end of word
- `3cW` = change 3 words
- `BcW` = to begin of full word, change full word
- `ciW` = change inner word.
- `ci"` = change inner between ".."
- `ci(` = change text between (..)
- `ci<` = change text between < .. > (needs `set matchpairs+=<:>` in vimrc)
- `4dd` = delete 4 lines
- `3x` = delete 3 characters.
- `3s` = substitute 3 characters.

Useful programmer commands

- `r` replace one character (e.g. `rd` replaces the current char with `d`).
- `~` changes case.
- `J` joins two lines
- Ctrl+A / Ctrl+X increments/decrements a number.
- `.` repeat last command (a simple macro)
- `==` fix line indent
- `>` indent block (in visual mode)
- `<` unindent block (in visual mode)

Macro recording

- Press `q[ key ]` to start recording.
- Then hit `q` to stop recording.
- The macro can be played with `@[ key ]` .

By using very specific commands and movements, VIM can replay those exact actions for the next lines. (e.g. A for append-to-end, `b` / `e` to move the cursor to the begin or end of a word respectively)

Example of well built settings

```
# reset to vim-defaults
if &compatible          # only if not set before:
  set nocompatible      # use vim-defaults instead of vi-defaults (easier, more
user friendly)
endif

# display settings
set background=dark     # enable for dark terminals
set nowrap              # dont wrap lines
set scrolloff=2         # 2 lines above/below cursor when scrolling
set number              # show line numbers
set showmatch           # show matching bracket (briefly jump)
set showmode            # show mode in status bar (insert/replace/...)
set showcmd             # show typed command in status bar
set ruler               # show cursor position in status bar
set title               # show file in titlebar
set wildmenu            # completion with menu
set wildignore=*.o,*.obj,*.bak,*.exe,*.py[co],*.swp,*~,*.pyc,.svn
set laststatus=2        # use 2 lines for the status bar
set matchtime=2         # show matching bracket for 0.2 seconds
set matchpairs+=<:>     # specially for html

# editor settings
set esckeys             # map missed escape sequences (enables keypad keys)
set ignorecase          # case insensitive searching
set smartcase           # but become case sensitive if you type uppercase
characters
set smartindent         # smart auto indenting
set smarttab            # smart tab handling for indenting
set magic               # change the way backslashes are used in search patterns
set bs=indent,eol,start # Allow backspacing over everything in insert mode

set tabstop=4           # number of spaces a tab counts for
set shiftwidth=4        # spaces for autoindents
#set expandtab           # turn a tabs into spaces

set fileformat=unix     # file mode is unix
#set fileformats=unix,dos    # only detect unix file format, displays that ^M with
dos files

# system settings
set lazyredraw          # no redraws in macros
set confirm             # get a dialog when :q, :w, or :wq fails
set nobackup            # no backup~ files.
set viminfo='20,\"500   # remember copy registers after quitting in the .viminfo
file -- 20 jump links, regs up to 500 lines'
set hidden              # remember undo after quitting
set history=50          # keep 50 lines of command history
set mouse=v             # use mouse in visual mode (not normal,insert,command,help
mode


# color settings (if terminal/gui supports it)
if &t_Co > 2 || has("gui_running")
  syntax on          # enable colors
  set hlsearch       # highlight search (very useful!)
  set incsearch      # search incremently (search while typing)
endif

# paste mode toggle (needed when using autoindent/smartindent)
map <F10> :set paste<CR>
map <F11> :set nopaste<CR>
imap <F10> <C-O>:set paste<CR>
imap <F11> <nop>
set pastetoggle=<F11>

# Use of the filetype plugins, auto completion and indentation support
filetype plugin indent on

# file type specific settings
if has("autocmd")
  # For debugging
  #set verbose=9

  # if bash is sh.
  let bash_is_sh=1

  # change to directory of current file automatically
  autocmd BufEnter * lcd %:p:h

  # Put these in an autocmd group, so that we can delete them easily.
  augroup mysettings
    au FileType xslt,xml,css,html,xhtml,javascript,sh,config,c,cpp,docbook set
smartindent shiftwidth=2 softtabstop=2 expandtab
    au FileType tex set wrap shiftwidth=2 softtabstop=2 expandtab

    # Confirm to PEP8
    au FileType python set tabstop=4 softtabstop=4 expandtab shiftwidth=4
cinwords=if,elif,else,for,while,try,except,finally,def,class
  augroup END

  augroup perl
    # reset (disable previous 'augroup perl' settings)
    au!

    au BufReadPre,BufNewFile
    \ *.pl,*.pm
    \ set formatoptions=croq smartindent shiftwidth=2 softtabstop=2 cindent
cinkeys='0{,0},!^F,o,O,e' " tags=./tags,tags,~/devel/tags,~/devel/C
    # formatoption:
    #   t - wrap text using textwidth
    #   c - wrap comments using textwidth (and auto insert comment leader)
    #   r - auto insert comment leader when pressing <return> in insert mode
    #   o - auto insert comment leader when pressing 'o' or 'O'.
    #   q - allow formatting of comments with "gq"
    #   a - auto formatting for paragraphs
    #   n - auto wrap numbered lists
    #
  augroup END


  # Always jump to the last known cursor position.
  # Don't do it when the position is invalid or when inside
  # an event handler (happens when dropping a file on gvim).
  autocmd BufReadPost *
    \ if line("'\"") > 0 && line("'\"") <= line("$") |
    \   exe "normal g`\"" |
    \ endif

endif # has("autocmd")
```

The settings can be stored in `~/.vimrc`, or system-wide in `/etc/vimrc.local` and then by read from the `/etc/vimrc` file using:

```
source /etc/vimrc.local
```

(you'll have to replace the `#` comment character with `"` to make it work in VIM, I wanted to give proper syntax highlighting here).

The commands I've listed here are pretty basic, and the main ones I use so far. They already make me quite more productive, without having to know all the fancy stuff.

edited Oct 29 '12 at 18:51 community wiki
12 revs, 3 users 99%
vdboor

2 OH GOD. Thank you for C-A and C-X – Metz Feb 13 '12 at 18:08

4 Better than `'.` is `g;`, which jumps back through the `changelist`. Goes to the last edited position, instead of last edited line – naught101 Apr 28 '12 at 2:09

The Control + R mechanism is very useful :-) In either **insert mode** or **command mode** (i.e. on the `:` line when typing commands), continue with a numbered or named register:

- a - z the named registers
- " the unnamed register, containing the text of the last delete or yank
- % the current file name
- # the alternate file name
- * the clipboard contents (X11: primary selection)
- + the clipboard contents
- / the last search pattern
- : the last command-line
- . the last inserted text
- - the last small (less than a line) delete
- =5*5 insert 25 into text (mini-calculator)

See `:help i_CTRL-R` and `:help c_CTRL-R` for more details, and snoop around nearby for more CTRL-R goodness.

edited Apr 12 '12 at 7:46 community wiki
5 revs, 4 users 53%
kev

10 FYI, this refers to Ctrl+R in *insert mode*. In normal mode, Ctrl+R is redo. – vdboor Jun 3 '10 at 9:08

2 +1 for current/alternate file name. `Control-A` also works in insert mode for last inserted text, and `Control-@` to both insert last inserted text and immediately switch to normal mode. – Aryeh Leib Taurog Feb 26 '12 at 19:06

Vim Plugins

There are a lot of good answers here, and one amazing one about the zen of vi. One thing I don't see mentioned is that vim is extremely extensible via plugins. There are scripts and plugins to make it do all kinds of crazy things the original author never considered. Here are a few examples of incredibly handy vim plugins:

rails.vim

Rails.vim is a plugin written by tpope. It's an incredible tool for people doing rails development. It does magical context-sensitive things that allow you to easily jump from a method in a controller to the associated view, over to a model, and down to unit tests for that model. It has saved dozens if not hundreds of hours as a rails developer.

gist.vim

This plugin allows you to select a region of text in visual mode and type a quick command to post it to gist.github.com. This allows for easy pastebin access, which is incredibly handy if you're collaborating with someone over IRC or IM.

space.vim

This plugin provides special functionality to the spacebar. It turns the spacebar into something analogous to the period, but instead of repeating actions it repeats motions. This can be very handy for moving quickly through a file in a way you define on the fly.

surround.vim

This plugin gives you the ability to work with text that is delimited in some fashion. It gives you objects which denote things inside of parens, things inside of quotes, etc. It can come in handy for manipulating delimited text.

supertab.vim

This script brings fancy tab completion functionality to vim. The autocomplete stuff is already there in the core of vim, but this brings it to a quick tab rather than multiple different multikey shortcuts. Very handy, and incredibly fun to use. While it's not VS's intellisense, it's a great step and brings a great deal of the functionality you'd like to expect from a tab completion tool.

syntastic.vim

This tool brings external syntax checking commands into vim. I haven't used it personally, but I've heard great things about it and the concept is hard to beat. Checking syntax without having to do it manually is a great time saver and can help you catch syntactic bugs as you introduce them rather than when you finally stop to test.

fugitive.vim

Direct access to git from inside of vim. Again, I haven't used this plugin, but I can see the utility. Unfortunately I'm in a culture where svn is considered "new", so I won't likely see git at work for quite some time.

nerdtree.vim

A tree browser for vim. I started using this recently, and it's really handy. It lets you put a treeview in a vertical split and open files easily. This is great for a project with a lot of source files you frequently jump between.

FuzzyFinderTextmate.vim

This is an unmaintained plugin, but still incredibly useful. It provides the ability to open files using a "fuzzy" descriptive syntax. It means that in a sparse tree of files you need only type enough characters to disambiguate the files you're interested in from the rest of the cruft.

Conclusion

There are a lot of incredible tools available for vim. I'm sure I've only scratched the surface here, and it's well worth searching for tools applicable to your domain. The combination of traditional vi's powerful toolset, vim's improvements on it, and plugins which extend vim even further, it's one of the most powerful ways to edit text ever conceived. Vim is easily as powerful as emacs, eclipse, visual studio, and textmate.

Thanks

Thanks to duwanis for his vim configs from which I have learned much and borrowed most of the plugins listed here.

answered Apr 1 '10 at 3:44 community wiki Benson

5 I think it's time FuzzyFinderTextmate started to get replaced with github.com/wincent/Command-T – Gavin Gilmour Jan 15 '11 at 13:44

2 +1 for Syntastic. That, combined with JSLint, has made my Javascript much less error-prone. See superuser.com/questions/247012/… about how to set up JSLint to work with Syntastic. – Nathan Long Mar 1 '11 at 19:07

1 @Benson Great list! I'd toss in snipMate as well. Very helpful automation of common coding stuff. if<tab> instant if block, etc. – AIG Sep 13 '11 at 17:37

2 I think nerdcommenter is also a good plugin: here. Like its name says, it is for commenting your code. – EarlOfEgo May 12 '12 at 15:13

`.` Repeat last text-changing command

I save a lot of time with this one.

Visual mode was mentioned previously, but block visual mode has saved me a lot of time when editing fixed size columns in text file. (accessed with Ctrl-V).

edited Mar 31 '10 at 23:01 community wiki 4 revs, 2 users 89% Cooper6581

3 Additionally, if you use a concise command (e.g. A for append-at-end) to edit the text, vim can repeat that exact same action for the next line you press the `.` key at. – vdboor Apr 1 '10 at 8:34

---

gi

Go to last edited location (very useful if you performed some searching and than want go back to edit)

^P and ^N

Complete previous (^P) or next (^N) text.

^O and ^I

Go to previous (`^O - "O"` for old) location or to the next (`^I - "I"` just near to `"O"`). When you perform searches, edit files etc., you can navigate through these "jumps" forward and back.

edited Dec 24 '12 at 14:50 community wiki
3 revs, 3 users 87%
dimba

6 Thanks for `gi` ! Now I don't need marks for that! – R. Martinho Fernandes Apr 1 '10 at 3:02

3 I Think this can also be done with `` – Kungi Feb 10 '11 at 16:23

4 @Kungi `. will take you to the last edit `` will take you back to the position you were in before the last 'jump' - which /might/ also be the position of the last edit. – Grant McLean Aug 23 '11 at 8:21

I recently (got) discovered this site: http://vimcasts.org/

It's pretty new and really really good. The guy who is running the site switched from textmate to vim and hosts very good and concise casts on specific vim topics. Check it out!

answered Mar 31 '10 at 19:37 community wiki
Ronny Brendel

3 If you like vim tutorials, check out Derek Wyatt's vim videos as well. They're excellent. – Jeromy Anglim Jan 13 '11 at 6:40

CTRL + A increments the number you are standing on.

edited Feb 27 '10 at 11:20 community wiki
2 revs, 2 users 67%
hcs42

19 ... and CTRL-X decrements. – innaM Aug 3 '09 at 9:14

9 It's a neat shortcut but so far I have NEVER found any use for it. – SolutionYogi Feb 26 '10 at 20:43

8 if you run vim in screen and wonder why this doesn't work - ctrl+A, A – matja Feb 27 '10 at 14:21

13 @SolutionYogi: Consider that you want to add line number to the beginning of each line. Solution: ggI1<space><esc>0qqyawjP0<c-a>0q9999@q – hcs42 Feb 27 '10 at 19:05

10 Extremely useful with Vimperator, where it increments (or decrements, Ctrl-X) the last number in the URL. Useful for quickly surfing through image galleries etc. – blueyed Apr 1 '10 at 14:47

All in Normal mode:

f<char> to move to the next instance of a particular character on the current line, and **;** to repeat.

F<char> to move to the previous instance of a particular character on the current line and **;** to repeat.

If used intelligently, the above two can make you killer-quick moving around in a line.

***** on a word to search for the next instance.

on a word to search for the previous instance.

edited Aug 28 '09 at 15:23 community wiki
3 revs
Eric Smith

6 Whoa, I didn't know about the * and # (search forward/back for word under cursor) binding. That's kinda cool. The f/F and t/T and ; commands are quick jumps to characters on the current line. f/F put the cursor on the indicated character while t/T puts it just up "to" the character (the character just before or after it according to the direction chosen. ; simply repeats the most recent f/F/t/T jump (in the same direction). – Jim Dennis Mar 14 '10 at 6:38

10 :) The tagline at the top of the tips page at vim.org: "Can you imagine how many keystrokes could have been saved, if I only had known the "*" command in time?" - Juergen Salk, 1/19/2001" – Steve K Apr 3 '10 at 23:50

1 As Jim mentioned, the "t/T" combo is often just as good, if not better, for example, `ct(` will erase the

word and put you in insert mode, but keep the parantheses! – puk Feb 24 '12 at 6:45

Session

a. save session

> :mks *sessionname*

b. force save session

> :mks! *sessionname*

c. load session

> gvim or vim -S *sessionname*

Adding and Subtracting

a. Adding and Subtracting

> CTRL-A ;Add [count] to the number or alphabetic character at or after the cursor. {not in Vi
>
> CTRL-X ;Subtract [count] from the number or alphabetic character at or after the cursor. {not in Vi}

b. Window key unmapping

> In window, Ctrl-A already mapped for whole file selection you need to unmap in rc file. mark mswin.vim CTRL-A mapping part as comment or add your rc file with unmap

c. With Macro

> The CTRL-A command is very useful in a macro. Example: Use the following steps to make a numbered list.
>
> 1. Create the first list entry, make sure it starts with a number.
> 2. qa - start recording into buffer 'a'
> 3. Y - yank the entry
> 4. p - put a copy of the entry below the first one
> 5. CTRL-A - increment the number
> 6. q - stop recording
> 7. @a - repeat the yank, put and increment times

answered Aug 19 '10 at 8:08 community wiki agfe2

1 Any idea what the shortcuts are in Windows? – Don Reba Aug 22 '10 at 5:22

Last week at work our project inherited a lot of Python code from another project. Unfortunately the code did not fit into our existing architecture - it was all done with global variables and functions, which would not work in a multi-threaded environment.

We had ~80 files that needed to be reworked to be object oriented - all the functions moved into classes, parameters changed, import statements added, etc. We had a list of about 20 types of fix that needed to be done to each file. I would estimate that doing it by hand one person could do maybe 2-4 per day.

So I did the first one by hand and then wrote a vim script to automate the changes. Most of it was a list of vim commands e.g.

```
" delete an un-needed function "
g/someFunction(/ d

" add wibble parameter to function foo "
%s/foo(/foo( wibble,/

" convert all function calls bar(thing) into method calls thing.bar() "
g/bar(/ normal nmaf(ldi(`aPa.
```

The last one deserves a bit of explanation:

```
g/bar(/  executes the following command on every line that contains "bar("
normal   execute the following text as if it was typed in in normal mode
n        goes to the next match of "bar(" (since the :g command leaves the cursor
position at the start of the line)
```

```
ma       saves the cursor position in mark a
f(       moves forward to the next opening bracket
l        moves right one character, so the cursor is now inside the brackets
di(      delete all the text inside the brackets
`a       go back to the position saved as mark a (i.e. the first character of
"bar")
P        paste the deleted text before the current cursor position
a.       go into insert mode and add a "."
```

For a couple of more complex transformations such as generating all the import statements I embedded some python into the vim script.

After a few hours of working on it I had a script that will do at least 95% of the conversion. I just open a file in vim then run `:source fixit.vim` and the file is transformed in a blink of the eye.

We still have the work of changing the remaining 5% that was not worth automating and of testing the results, but by spending a day writing this script I estimate we have saved weeks of work.

Of course it would have been possible to automate this with a scripting language like Python or Ruby, but it would have taken far longer to write and would be less flexible - the last example would have been difficult since regex alone would not be able to handle nested brackets, e.g. to convert `bar(foo(xxx))` to `foo(xxx).bar()`. Vim was perfect for the task.

edited Aug 18 '12 at 21:44 community wiki
8 revs, 2 users 98%
Dave Kirby

1 Thanks a lot for sharing that's really nice to learn from "useful & not classical" macros. – Olivier Pons Feb 28 '10 at 14:41

1 Of if you don't like vim-style escapes, use \v to turn on Very Magic: `%s/\v(bar)\((.+)\)/\2.\1()/` – Ipsquiggle Mar 23 '10 at 16:56

Use the builtin file explorer! The command is `:Explore` and it allows you to navigate through your source code very very fast. I have these mapping in my `.vimrc`:

```
map <silent> <F8>   :Explore<CR>
map <silent> <S-F8> :sp +Explore<CR>
```

The explorer allows you to make file modifications, too. I'll post some of my favorite keys, pressing `<F1>` will give you the full list:

- **-**: The most useful: Change to upper directory (`cd ..`)
- **mf**: Mark a file
- **D**: Delete marked files or the file the cursor is on, if nothing ismarked.
- **R**: Rename the file the cursor is on.
- **d**: Create a new directory in the current directory
- **%**: Create a new file in the current directory

edited Aug 2 '09 at 11:17 community wiki
2 revs
soulmerge

1 I always thought the default methods for browsing kinda sucked for most stuff. It's just slow to browse, if you know where you wanna go. LustyExplorer from vim.org's script section is a much needed improvement. – Svend Aug 2 '09 at 8:48

8 I recommend NERDtree instead of the built-in explorer. It has changed the way I used vim for projects and made me much more productive. Just google for it. – kprobst Apr 1 '10 at 3:53

I am a member of the American Cryptogram Association. The bimonthly magazine includes over 100 cryptograms of various sorts. Roughly 15 of these are "cryptarithms" - various types of arithmetic problems with letters substituted for the digits. Two or three of these are sudokus, except with letters instead of numbers. When the grid is completed, the nine distinct letters will spell out a word or words, on some line, diagonal, spiral, etc., somewhere in the grid.

Rather than working with pencil, or typing the problems in by hand, I download the problems from the members area of their website.

When working with these sudokus, I use vi, simply because I'm using facilities that vi has that few other editors have. Mostly in converting the lettered grid into a numbered grid, because I find it easier to solve, and then the completed numbered grid back into the lettered grid to find the solution word or words.

The problem is formatted as nine groups of nine letters, with `-` s representing the blanks, written in two lines. The first step is to format these into nine lines of nine characters each. There's nothing special about this, just inserting eight linebreaks in the appropriate places.

The result will look like this:

```
T-O-----C
-E-----S-
--AT--N-L
---NASO--
---E-T---
--SPCL---
E-T--OS--
-A-----P-
S-----C-T
```

So, first step in converting this into numbers is to make a list of the distinct letters. First, I make a copy of the block. I position the cursor at the top of the block, then type `:y}}p` . `:` puts me in command mode, `y` yanks the next movement command. Since `}` is a move to the end of the next paragraph, `y}` yanks the paragraph. `}` then moves the cursor to the end of the paragraph, and `p` pastes what we had yanked just after the cursor. So `y}}p` creates a copy of the next paragraph, and ends up with the cursor between the two copies.

Next, I to turn one of those copies into a list of distinct letters. That command is a bit more complex:

```
:!}tr -cd A-Z | sed 's/\(.\)/\1\n/g' | sort -u | tr -d '\n'
```

`:` again puts me in command mode. `!` indicates that the content of the next yank should be piped through a command line. `}` yanks the next paragraph, and the command line then uses the `tr` command to strip out everything except for upper-case letters, the `sed` command to print each letter on a single line, and the `sort` command to sort those lines, removing duplicates, and then `tr` strips out the newlines, leaving the nine distinct letters in a single line, replacing the nine lines that had made up the paragraph originally. In this case, the letters are: `ACELNOPST` .

Next step is to make another copy of the grid. And then to use the letters I've just identified to replace each of those letters with a digit from 1 to 9. That's simple: `:!}tr ACELNOPST 0-9` . The result is:

```
8-5-----1
-2-----7-
--08--4-3
---4075--
---2-8---
--7613---
2-8--57--
-0-----6-
7-----1-8
```

This can then be solved in the usual way, or entered into any sudoku solver you might prefer. The completed solution can then be converted back into letters with `:!}tr 1-9 ACELNOPST` .

There is power in vi that is matched by very few others. The biggest problem is that only a very few of the vi tutorial books, websites, help-files, etc., do more than barely touch the surface of what is possible.

edited Jun 15 '11 at 13:39 community wiki
2 revs, 2 users 92%
Jeff Dege

1 Doesn't vim check the name it was started with so that it can come up in the right 'mode'? – dash-tom-bang Aug 24 '11 at 0:45

3 I'm baffled by this repeated error: you say you need `:` to go into command mode, but then invariably you specify *normal mode* commands (like `y}}p`) which cannot possibly work from the command mode?! – sehe Mar 4 '12 at 20:47

Bulk text manipulations!

Either through macros:

- Start with recording: `qq`
- Do stuff
- Stop recording: `q`
- Repeat: `@q` (the first time), `@@` after that.
- Repeat 20 times: `20@@`

Or through regular expressions:

- Replace stuff: `:%s/[fo]+/bar/g`

(But be warned: if you do the latter, you'll have 2 problems :).)

answered Aug 2 '09 at 8:59 community wiki
jqno

5 +1 for the Jamie Zawinski reference. (No points taken back for failing to link to it, even). :) – Jim Dennis Jan 10 '10 at 4:03

1 @Jim I didn't even know it was a Jamie Zawinski quote :). I'll try to remember it from now on. – jqno Jan 10 '10 at 10:06

I recently discovered `q:` . It opens the "command window" and shows your most recent ex-mode (command-mode) commands. You can move as usual within the window, and pressing `<CR>` executes the command. You can edit, etc. too. Priceless when you're messing around with some complex command or regex and you don't want to retype the whole thing, or if the complex thing you want to do was 3 commands back. It's almost like bash's `set -o vi` , but for vim itself (heh!).

See `:help q:` for more interesting bits for going back and forth.

answered Apr 2 '12 at 7:56 community wiki
David Pope

I just discovered Vim's omnicompletion the other day, and while I'll admit I'm a bit hazy on what does which, I've had surprisingly good results just mashing either Ctrl + x Ctrl + u or Ctrl + n / Ctrl + p in insert mode. It's not quite IntelliSense, but I'm still learning it.

Try it out! `:help ins-completion`

edited Feb 27 '10 at 11:29 community wiki
2 revs, 2 users 56%
Peter Mortensen

These are not shortcuts, but they are related:

1. Make capslock an additional ESC (or Ctrl)
2. map leader to "," (comma), with this command: let mapleader=","

They boost my productivity.

answered Mar 14 '10 at 19:49 community wiki
tfmoraes

2 To make Caps Lock an additional Esc in Windows (what's a caps lock key for? An "any key"?), try this: webpages.charter.net/krumsick – R. Martinho Fernandes Apr 1 '10 at 3:30

1 On Windows I use AutoHotKey with `Capslock::Escape` – Jeromy Anglim Jan 10 '11 at 4:43

Another useful vi "shortcut" I frequently use is 'xp'. This will swap the character under the cursor with the next character.

answered Sep 20 '10 at 10:34 community wiki
Costyn

3 `Xp` to go the other way – tester Aug 22 '11 at 17:19

5 Around the time that Windows xp came out, I used to joke that this is the only good use for it. – kguest Aug 27 '11 at 8:21

<Ctrl> + W, V to split the screen vertically
<Ctrl> + W, W to shift between the windows

!python % [args] to run the script I am editing in this window

ZF in visual mode to fold arbitrary lines

answered Aug 2 '09 at 9:47 community wiki
Peter Ellis

2 <Ctrl> + W and j/k will let you navigate absolutely (j up, k down, as with normal vim). This is great when you have 3+ splits. – Andrew Scagnelli Apr 1 '10 at 2:58

1 +1 for zf in visual mode, I like code folding, but did not know about that. – coder_tim Jan 30 '12 at 20:08

2 after bashing my keyboard I have deduced that `<C-w>n` or `<C-w>s` is new horizontal window, `<C-w>b` is bottom right window, `<C-w>c` or `<C-w>q` is close window, `<C-w>x` is increase and then decrease window width (??), `<C-w>p` is last window, `<C-w>backspace` is move left(ish) window – puk Feb 24 '12 at 7:00

Visual Mode

As several other people have said, visual mode is the answer to your copy/cut & paste problem. Vim gives you 'v', 'V', and C-v. Lower case 'v' in vim is essentially the same as the shift key in notepad. The nice thing is that you don't have to hold it down. You can use any movement technique to navigate efficiently to the starting (or ending) point of your selection. Then hit 'v', and use efficient movement techniques again to navigate to the other end of your selection. Then 'd' or 'y' allows you to cut or copy that selection.

The advantage vim's visual mode has over Jim Dennis's description of cut/copy/paste in vi is that you don't have to get the location exactly right. Sometimes it's more efficient to use a quick movement to get to the general vicinity of where you want to go and then refine that with other movements than to think up a more complex single movement command that gets you exactly where you want to go.

The downside to using visual mode extensively in this manner is that it can become a crutch that you use all the time which prevents you from learning new vi(m) commands that might allow you to do things more efficiently. However, if you are very proactive about learning new aspects of vi(m), then this probably won't affect you much.

I'll also re-emphasize that the visual line and visual block modes give you variations on this same theme that can be very powerful...especially the visual block mode.

On Efficient Use of the Keyboard

I also disagree with your assertion that alternating hands is the fastest way to use the keyboard. It has an element of truth in it. Speaking very generally, repeated use of the same thing is slow. This most significant example of this principle is that consecutive keystrokes typed with the same finger are very slow. Your assertion probably stems from the natural tendency to use the s/finger/hand/ transformation on this pattern. To some extent it's correct, but at the extremely high end of the efficiency spectrum it's incorrect.

Just ask any pianist. Ask them whether it's faster to play a succession of a few notes alternating hands or using consecutive fingers of a single hand in sequence. The fastest way to type 4 keystrokes is not to alternate hands, but to type them with 4 fingers of the same hand in either ascending or descending order (call this a "run"). This should be self-evident once you've considered this possibility.

The more difficult problem is optimizing for this. It's pretty easy to optimize for absolute distance on the keyboard. Vim does that. It's much harder to optimize at the "run" level, but vi(m) with it's modal editing gives you a better chance at being able to do it than any non-modal approach (ahem, emacs) ever could.

On Emacs

Lest the emacs zealots completely disregard my whole post on account of that last parenthetical comment, I feel I must describe the root of the difference between the emacs and vim religions. I've never spoken up in the editor wars and I probably won't do it again, but I've never heard anyone describe the differences this way, so here it goes. The difference is the following tradeoff:

Vim gives you unmatched raw text editing efficiency Emacs gives you unmatched ability to customize and program the editor

The blind vim zealots will claim that vim has a scripting language. But it's an obscure, ad-hoc language that was designed to serve the editor. Emacs has Lisp! Enough said. If you don't appreciate the significance of those last two sentences or have a desire to learn enough about functional programming and Lisp to develop that appreciation, then you should use vim.

The emacs zealots will claim that emacs has viper mode, and so it is a superset of vim. But viper mode isn't standard. My understanding is that viper mode is not used by the majority of emacs users. Since it's not the default, most emacs users probably don't develop a true appreciation for the benefits of the modal paradigm.

In my opinion these differences are orthogonal. I believe the benefits of vim and emacs as I have stated them are both valid. This means that the ultimate editor doesn't exist yet. It's probably true that emacs would be the easiest platform on which to base the ultimate editor. But modal editing is not entrenched in the emacs mindset. The emacs community could move that way in the future, but that doesn't seem very likely.

So if you want raw editing efficiency, use vim. If you want the ultimate environment for scripting and programming your editor use emacs. If you want some of both with an emphasis on programmability, use emacs with viper mode (or program your own mode). If you want the best of both worlds, you're out of luck for now.

edited Apr 1 '10 at 17:00

community wiki
2 revs
user307058

Spend 30 mins doing the vim tutorial (run vimtutor instead of vim in terminal). You will learn the basic movements, and some keystrokes, this will make you at least as productive with vim as with the text editor you used before. After that, well, read Jim Dennis' answer again :)

answered Mar 31 '10 at 22:44
community wiki
konryd

1 This is the first thing I thought of when reading the OP. It's obvious that the poster has never run this; I ran through it when first learning vim two years ago and it cemented in my mind the superiority of Vim to any of the other editors I've used (including, for me, Emacs since the key combos are annoying to use on a Mac). – dash-tom-bang Aug 24 '11 at 0:47

> What is the way you use Vim that makes you more productive than with a contemporary editor?

Being able to execute complex, repetitive edits with very few keystrokes (often using macros). Take a look at VimGolf to witness the power of Vim!

After over ten years of almost daily usage, it's hard to imagine using any other editor.

answered Jan 12 '11 at 22:52
community wiki
Johnsyweb

Use `\c` anywhere in a search to ignore case (overriding your ignorecase or smartcase settings). E.g. `/\cfoo` or `/foo\c` will match `foo` , `Foo` , `fOO` , `FOO` , etc.

Use `\C` anywhere in a search to force case matching. E.g. `/\Cfoo` or `/foo\C` will only match foo.

edited Jun 15 '11 at 13:42
community wiki
2 revs, 2 users 67%
kev

I was surprised to find no one mention the `t` movement. I frequently use it with parameter lists in the form of `dt,` or `yt,`

edited Jun 15 '11 at 13:44
community wiki
2 revs, 2 users 67%
David Corbin

2 or dfx, dFx, dtx, ytx, etc where x is a char, +1 – hhh Jan 14 '11 at 17:09

Odd nobody's mentioned ctags. Download "exuberant ctags" and put it ahead of the crappy preinstalled version you already have in your search path. Cd to the root of whatever you're working on; for example the Android kernel distribution. Type "ctags -R ." to build an index of source files anywhere beneath that dir in a file named "tags". This contains all tags, nomatter the language nor where in the dir, in one file, so cross-language work is easy.

Then open vim in that folder and read :help ctags for some commands. A few I use often:

- Put cursor on a method call and type CTRL-] to go to the method definition.
- Type :ta name to go to the definition of name.

edited May 6 '12 at 20:50
community wiki
3 revs
Bradjcox

Automatic indentation:

`gg` (go to start of document)
`=` (indent time!)
`shift-g` (go to end of document)

You'll need 'filetype plugin indent on' in your `.vimrc` file, and probably appropriate 'shiftwidth' and 'expandtab' settings.

edited Feb 27 '10 at 11:19
community wiki
2 revs, 2 users 67%
David Claridge

4 Or just use the ":set ai" (auto-indent) facility, which has been in vi since the beginning. – xcramps Aug 28 '09 at 17:14

You asked about productive shortcuts, but I think your real question is: Is vim worth it? The answer to this stackoverflow question is -> "Yes"

You must have noticed two things. Vim is powerful, and vim is hard to learn. Much of it's power lies in it's expandability and endless combination of commands. Don't feel overwhelmed. Go slow. One command, one plugin at a time. Don't overdo it.

All that investment you put into vim will pay back a thousand fold. You're going to be inside a text editor for many, many hours before you die. Vim will be your companion.

answered Jul 24 '10 at 5:41 community wiki
autodidakto

Multiple buffers, and in particular fast jumping between them to compare two files with `:bp` and `:bn` (properly remapped to a single Shift + p or Shift + n)

`vimdiff` mode (splits in two vertical buffers, with colors to show the differences)

Area-copy with Ctrl + v

And finally, tab completion of identifiers (search for "mosh_tab_or_complete"). That's a life changer.

edited Feb 27 '10 at 11:23 community wiki
2 revs, 2 users 67%
Stefano Borini

Agreed with the top poster - the **:r!** command is *very* useful.

Most often I use it to "paste" things:

```
:r!cat
**Ctrl-V to paste from the OS clipboard**
^D
```

This way I don't have to fiddle with `:set paste`.

answered Aug 28 '09 at 16:07 community wiki
David Wolever

3 Probably better to set the `clipboard` option to `unnamed` (`set clipboard=unnamed` in your .vimrc) to use the system clipboard by default. Or if you still want the system clipboard separate from the unnamed register, use the appropriately named clipboard register: `"*p` . – R. Martinho Fernandes Apr 1 '10 at 3:17

2 Love it! After being exasperated by pasting code examples from the web and I was just starting to feel proficient in vim. That was the command I dreamed up on the spot. This was when vim totally hooked me. – kevpie Oct 12 '10 at 22:38

1 2 next