Power *Vim* Usage

Smylers
Smylers@stripey.com
Donhost

UKUUG Linux 2004 Conference • 2004 August
http://www.ukuug.org/events/linux2004/

1 Intro

- text editing:
  - crops up all over Linux
  - once good at it, find many uses for it
- *Vim*:
  - powerful — potential for great efficiency
  - *Emacs* and other *VI* variants also good
- this talk:
  - tips — features & customizations
  - fast, random, dull
  - to encourage reading the notes
  - work in progress

Presenter Notes

- more you do, more you want to do — prefer to use text editing when there's a choice of ways of doing things:
  - trainer for GBdirect, teaching Linux, PHP, *Apache*, yet *Vim* cropping up
  - learning a good editor one of best efficiency gains — perhaps up with touch-typing
  - config files in `/etc/` rather than gui admin tool
  - e-mails
  - HTML, PostScript, *OpenOffice*, *XFig* files
  - *Vim* simply cos I happen to use it
  - if just here for a holy war, not going to get it, so any *Emacs* users in the room can leave now
- Unix-based

2 Learning *Vim*

- *Vim* is far too big to learn quickly
- probably too big to learn slowly
- learning it as an extension of *VI* is often not helpful

Presenter Notes

- I don't know all of it

The only way to get the hang of *Vim* is a bit at a time — learning some things you can make use of, getting comfortable with them, then learning some more.

Some of the most useful *Vim* features don't require any *VI* knowledge. Much *VI* stuff is actually easier to pick up using *Vim* techniques.

3 Moving About

- navigating in a file is basic and essential
- incremental searching:

  `:set incsearch`
- case-insensitive searching by default:

  `:set ignorecase smartcase`
- jumping back to where you've been: `Ctrl+O`, `Ctrl+I`
- moving along a line: `f`, `t`, `;`
- text just changed: `'[`, `']`, `'<`
- start of block: `[{`

Presenter Notes

- how much time spent just moving about?
- `%` well known, but `[{` not so — not for Python, though
- see not so hard — guessed two commands already

The slides use options' long names for clarity, but they all have shorter forms that can be used when typing them, such as `ic` for `ignorecase` and `scs` for `smartcase`. Also, remember that *Vim* has tab completion for typing things such as option names and help topics.

`incsearch` makes searching considerably easier, because you can stop typing (and just press `Enter`) as soon as you see that you're where you want to be.

Having searches be case-insensitive is a good default; the presence of any capital letters in the pattern triggers case-sensitive matching. Or put `\C` in the pattern to make a lower-case string case-sensitive.

To jump to somewhere else in a file use a search and go straight there rather than pressing `Down` till you see it. This also means that you can use `Ctrl+O` to jump back to where you were (much easier than the *VI* way with marks, because it doesn't require you to remember to set a mark before jumping). `Ctrl+O` and `Ctrl+I` take you up and down the list of recent positions; I remember the keys by thinking of them moving 'outwards' and 'inwards'.

`f` and `t` move you along the current line — better than pressing `Right` lots if you can see where you want to be.

Learning a few (more) movement commands helps with editing too, since so many commands such as `d`, `c`, and `v` can be followed by any movement command. For example type `df>` to delete up to the next right angle bracket, or `ct"` to change the text before the next quotes.

Capital `F` is similar to `f` but moves left instead of right. Guess what the left-moving variant of `t` is? See, once you know a few commands it becomes possible to infer others.

`'[` and `']` move to either end of the chunk of text most recently modified (inserted, indented, whatever). `'<` moves to the start of the most recent selection; guess what moves to the end. These are useful for performing several operations on the same text.

`[{` and friends is great if working in a language such as C or Perl with delimited blocks; it doesn't just search for the preceding left brace, but for the one that starts the block you're in — handy for discovering which function or `if` clause this is.

4 Visual Mode

- visual mode more intuitive than traditional *VI* ways
- `v`, `V`, `Ctrl+V`
- safety net when learning
  - `vt"c` rather than `ct"`
- counts can be inaccurate
  - `6dw` risky
- useful if boundary needs a little tweaking
  - `V}kd` deletes not quite to the end of the paragraph

Presenter Notes

- verb at the end: convenient if you speak German

This is a good example of a *Vim* feature making *VI* keystrokes easier to learn: trying out the various movement commands in visual mode is completely safe, then you can press `d` or `c` or whatever to act once the correct text has been highlighted.

5 Settings in *.vimrc*

- per-user settings in *.vimrc*
- many good features not enabled by default

```
set ignorecase smartcase
set incsearch
```

- disabling per-system settings

```
set nohlsearch
```

- worth spending some time crafting *.vimrc*

Presenter Notes

- most people aware of this
- any *Ex*-style colon command, without the colon
- much of the rest of this talk examples of things to put in there

Many excellent *Vim* features are hidden by default so that *Vim* behaves in a *VI*-compatible way and doesn't confuse traditional *VI* users. But then it ships with a default system-wide config file that enables some *Vim*-specific features (such as the search highlighting that irritates so many people), which undermines the compatibility without provding maximum usability.

6 Indenting

- indenting by hand
  - tedious
- automatic indenting
  - can be even worse
  - lovely when done right
- tabs
  - tab character (0x08): bad
  - `Tab` key: good

Indenting when done correctly is one of those things that makes an editor a pleasure to use, automating a very tedious job. When left to indent manually many people don't bother, or end up with an incoherent mess, a mixture of spaces and tabs and things that don't line up right.

This is one of the things that *Vim* is terrible at by default — often you seem to be working against the editor rather than with it — which seems even more of a shame once you finally discover that with only a few settings things can be so much better.

7 Indenting Done Right

- options:

```
set tabstop=8
set expandtab
set shiftwidth=2 " or whatever
set smarttab autoindent
```

- insert-mode keystrokes:
  - adjust indent: `Tab` and `BkSpc` at start of line
  - adjust indent: `Ctrl+T` and `Ctrl+D`
  - tab character: `Ctrl+V Tab`
- normal-mode keystrokes:
  - adjust indent: `>` and `<`
  - re-indent: `=`

Presenter Notes

- "smartcase", "smartindent" — possibly "smart" is *Vim*-speak for the way that it should've been done in the first place?

Highlighting a block of code and pressing `=` to reformat it is one of those things that can really impress your colleagues — especially if they've been struggling to do it by hand. `:retab` is a command worth knowing for changing tabs to spaces; you may need to adjust `tabstop` so that code looks 'right' first.

Some people dislike `autoindent`; if that's you then don't use it, but the other options are still worth having.

8 Pasting

- indented text pastes badly when running *Vim* in a terminal
- help *Vim* out:

```
nnoremap <F5> :set invpaste paste?<Enter>
imap <F5> <C-O><F5>
set pastetoggle=<F5>
```

  - `F5` before and after pasting

Presenter Notes

- kind of thing that makes many people give up, and go back to a simpler editor or retype stuff instead of pasting it in

This is something that many people find annoying; the behaviour can be quite puzzling until you realize what's going on. It's one of the things that gives automatic indenting a bad name (even though it isn't really *Vim*'s fault). Again *Vim* has a solution, but there isn't a keystroke mapped to it by default.

In a terminal when you've got `autoindent` turned on when you press `Enter` after an indented line *Vim* automatically adds some spaces at the beginning of that line. As a human being you see those spaces and don't type any yourself, unless you want to increase the indent.

The problem is that when you're pasting in already-indented text *Vim* still puts these default spaces in, but then some additional spaces from the pasted text get 'typed' in at the start of each line, increasing the indent on every line and leading to the 'stepped' effect down the page. The way that terminals work, *Vim* has no way of distinguishing text being pasted in from somebody just typing very fast.

This problem does not occur when using *Vim*'s own gui (started with the `gvim` command, for example), where *Vim* has full control of the input system and can spot when pasting is taking place.

9 Filetype-Dependent Settings

- different types of files need different settings
- historically awkward in *Vim*
- *Vim 6* finally got this right
- many handy settings just waiting to be activated

```
filetype plugin on
filetype indent on
```

- major usability gains

Wanting a different set-up for different types of files is a fairly basic requirement. Without this you're often compromising the default set-up, and often putting up with inconvenient settings. For example, generally tab characters are evil, but in makefiles they're a necessary evil so you don't want `expandtab` enabled when editing makefiles.

One of the problems is that because *Vim* made this so hard for so long, many people are not aware of how simple it can be now. Another problem is that for backwards compatibility *Vim* still accepts the older ways of attempting to do this, which makes the documentation much more complicated — with the result that even after reading the built-in help, many people are still not aware of how simple this now is.

But the few people that did work it out went ahead and customized their environments for different types of files. A bunch of these plug-ins are shipped with *Vim* ready for use — they just aren't enabled by default (is this sounding familiar?). So you can very easily take advantage of others' work in setting up per-filetype settings without needing to know how to do this yourself.

10 Custom Filetype Settings

- creating a filetype-specific setting requires knowing the filetype:

```
:set ft?
filetype=sql
```

- separate file, such as *~/.vim/ftplugin/sql.vim*
- options specified with `setlocal`

```
setlocal comments+=:--
```

Presenter Notes

- actually doesn't *require* knowing the filetype: you can often guess correctly

The first major feature *Vim 6* has for filetype-specific settings is using completely separate file for each filetype's settings. You just have to save the settings in a suitably named file and *Vim* loads them at the appropriate time. The example above specifies that double-hyphens can be used to denote single-line comments in SQL files.

If you create a plug-in file such as *~/.vim/ftplugin/perl.vim* for a filetype which already has a system-wide plug-in then yours will be used instead. Alternatively you can put your options in a file under *~/.vim/after/*, such as *~/.vim/after/ftplugin/perl.vim* and then the system-wide file will be run first and then your file afterwards.

11 Easier Typing

- same words crop up often
- `Ctrl+P` and `Ctrl+N` in insert mode
  - like `Tab`-completion of filenames
  - jargon & product names
  - long variable and function names
- cycling
- copying more: `Ctrl+X Ctrl+P`, `Ctrl+X Ctrl+L`
- copying above characters: `Ctrl+Y`
- filename completion: `Ctrl+X Ctrl+F`

Presenter Notes

- as if by magic example
- so used to it that want it elsewhere -- sometimes type half a word then press Ctrl+P, only to realize that not in *Vim*; do it in *Word* and the printer starts whirring

`Ctrl+P` is arguably *the* thing that's worth learning for impressing people looking over your shoulder: they can see words magically appearing on screen faster than you can type.

12 Spelling Help

- insert words straight from a dictionary

```
set dictionary=~/.ispell_british,/usr/share/dict/words
set complete=.,w,k
```

  - `Ctrl+N` and `Ctrl+P` in insert mode
  - also for long words
- check meanings

```
set keywordprg=dict
```

  - `K`

Presenter Notes

- pretend you can spell

Inserting correctly spelt words is better than having to check them later. Even when you can spell, only having to type the first few letters of a long word then press `Ctrl+N` a few times to fill in the rest can often be easier than typing it out.

13 Working with Text

- *Vim* obviously designed for coding
- very good with human text too
- *~/.vim/filetype.vim*:

```
if exists('did_load_filetypes')
  finish
endif
augroup filetypedetect
  autocmd! BufRead,BufNewFile *.txt setfiletype human
augroup END
```

- *~/.vim/ftplugin/human.vim*:

```
setlocal textwidth=79 formatoptions+=t
setlocal infercase
```

For some time after becoming a fan of *Vim* for editing code, I was sceptical of its utility in writing plain, human-readable text files; they don't seem to require many of *Vim*'s features, and some actually get in the way. Actually *Vim* is very good for dealing with text, and once you're familiar with it can be used much more efficiently than simpler editors.

However working comfortably with text requires different options being set. This is most easily achieved by creating a filetype for such files. Then you can customize the environment like for any other filetype; the above is just an example of a few options that are useful.

Vim already has a `mail` filetype, which is likely to want your text settings too. You can achieve this by starting *mail.vim* with this command:

```
runtime! human.vim
```

Further settings are added afterwards, and can of course override import settings. For example you probably want a narrower `textwidth` on e-mails.

You can use your *filetype.vim* to define a custom filetype for anything that *Vim* doesn't already support, or to persuade a non-standard extension to be treated as one of the existing filetypes.

14 Spelling Checking

- various ways of plumbing external spelling-checkers into *Vim*
- can have a keystroke for highlighting non-words in red:

```
nnoremap <buffer>
\ <F9> :silent call <SID>HighlightSpellingErrors()<Enter>
nnoremap <buffer>
\ <F10> :syntax clear SpellingError<Enter>

nnoremap <buffer>
\ <F8> :silent call <SID>AddWordToDictionary()<Enter>
```

- function definitions in the accompanying notes

Presenter Notes

- “spell-checking” something Professor McGonagall would do, saying the right magic words and so on
- other ways possible

To set up this style of spelling-checking, include all of the following in your *~/.vim/ftplugin/human.vim* file. You need to have *ISpell* installed, and may need to alter the global variables at the start.

```
nnoremap <F9> :silent call HighlightSpellingErrors()<Enter>
nnoremap <F10> :syntax clear SpellingError<Enter>

nnoremap <F8> :silent call AddWordToDictionary()<Enter>

" global variables used in these functions (PersonalDict is also handy for
" setting the complete option):
let ISpellLang = 'british'
let PersonalDict = '~/.ispell_' . ISpellLang

function! s:HighlightSpellingErrors()
" highlights spelling errors in the current window

  " Remove any previously-identified spelling errors (and corrections):
  silent! syntax clear SpellingError
  silent! syntax clear Normal

  " Pipe buffer contents  through 'ISpell' to generate a list of misspelt words
  " and store it in a temporary file:
  let ErrorsFile = tempname()
  let SpellerCmd = 'ispell -l -d ' . g:ISpellLang
  execute 'w ! ' . SpellerCmd . ' | sort -u > ' . ErrorsFile

  " Open that list of words in another window:
  execute 'split ' . ErrorsFile

  " For every word in that list ending with "'s", check if the root form
  " without the "'s" is in the dictionary, and if so remove the word from the
  " list:
  g /'s$/ execute 'r ! echo ' . expand('<cword>') . ' | ' . SpellerCmd | delete
  " (If the root form is in the dictionary, ispell -l will have no output so
  " nothing will be read in, the cursor will remain in the same place and the
  " :delete will delete the word from the list.  If the root form is not in the
  " dictionary, then ispell -l will output it and it will be read on to a new
  " line; the delete command will then remove that misspelt root form, leaving
  " the original possessive form in the list!)

  " Turn each mistake into a 'Vim' command to place it in the SpellingError
  " syntax highlighting group:
```

```
  %s/.*/syntax match SpellingError #\\v<&>#/

  " Save and close that file (so switch back to the one being checked):
  x

  " Make syntax highlighting case-sensitive, then execute all the match
  " commands that have just been set up in that temporary file, delete it, and
  " highlight all those words in red:
  syntax case match
  execute 'source ' . ErrorsFile
  call delete(ErrorsFile)
  highlight SpellingError term=reverse ctermfg=DarkRed guifg=Red

endfunction " HighlightSpellingErrors()


function! s:AddWordToDictionary()
" adds the word under the cursor to the personal dictonary

  " Get the word under the cursor, including the apostrophe as a word character
  " to allow for words like "won't", but then ignoring any apostrophes at the
  " start or end of the word:
  setlocal iskeyword+='
  let Word = substitute(expand('<cword>'), "^'\\+", '', '')
  let Word = substitute(Word, "'\\+$", '', '')
  setlocal iskeyword-='

  " Override any SpellingError highlighting that might exist for this word,
  " 'highlighting' it as normal text:
  execute 'syntax match Normal #\v<' . Word . '>#'

  " Remove any final "'s" so that possessive forms don't end up in the
  " dictionary, then add the word to the dictionary:
  let Word = substitute(Word, "'s$", '', '')
  execute '!echo "' . Word . '" >> ' . g:PersonalDict

endfunction " AddWordToDictionary()
```

15 Reflowing Text

- reflow a para with `Q`:

  ```
  nnoremap Q gqap
  ```

- reflow visually highlighted lines with `Q`:

  ```
  vnoremap Q gq
  ```

With the above mapping you can press `Q` every time you alter something in a paragraph, and the line-breaks get sorted out. The default meaning of `Q` is not useful, and anyway `gQ` makes a better job of it.

16 *Vim* in *Mozilla*

- frustrating not to have *Vim* everywhere
- can use it for editing text areas and viewing source in *Mozilla* and *Firefox*
- MozEX extension
  - commands something like `/usr/bin/X11/gvim %t`
  - on right-click submenu

It would be nice if this were integrated slightly more, but it's better than nothing.

Presenter Notes

- for years was still occasionally using *Lynx* on pages with textareas, just so could activate *Vim* on them.

17 CVS

- CVS more convenient when invoked directly from editor
- `\cd` for `diff` current file, in split window:

```
nnoremap \cd :silent call <SID>CVSDiff()<Enter>
function! s:CVSDiff()
  new
  setlocal buftype=nofile
  r ! cvs diff #
  1 delete
  filetype detect
endfunction
```

- `\cc` for `commit` current file:

```
nnoremap \cc :!cvs commit %<Enter>
```

Putting interaction with commonly used external programs in the editor greatly reduces the hassle of dealing with them. The above mappings are just examples of things you can do.

The `%` in the above `cvs commit` command gets replaced with the current filename. Using this mapping reduces the chance of committing a whole directory when you only wanted to commit one file. CVS will invoke `$EDITOR` for typing the commit message. Note that *Vim*'s gui doesn't emulate a terminal well enough to run *Vim* inside it. It's a good idea to put the following setting in *~/.gvimrc* to prevent this from happening:

```
let $EDITOR = 'gvim -f'
```


18 Working with Multiple Files

- opening lots of files at once:

```
% vim *html
% find -type f | xargs -o vim
% vim `find -type f`
```

- rapidly flicking through them with `Ctrl+N` and `Ctrl+P`:

```
nnoremap <C-N> :next<Enter>
nnoremap <C-P> :prev<Enter>
set confirm
```

- multiple windows:

```
% vim -o *html
```

Often people have a bunch of files to work with at once. There are several ways of specifying them to *Vim*, but switching between them is a tad tedious. Fortunately the keystrokes `Ctrl+N` and `Ctrl+P` don't do anything useful by default (they duplicate `j` and `k`), so they can be co-opted for something more useful.

`confirm` is another one of those options that almost everybody wants all the time, and yet it's beginners who have to make do without it. If you try to close a file with unsaved changes you probably wish to save them first; you may also wish to abandon the changes; but it's much less likely that you suddenly decide not to close that file at all.

By default *Vim* presumes the latter case, aborting your action with zero further keystrokes from you. Saving the changes involves about seven keystrokes (`:w<Enter>:<Up><Up><Enter>`); abandoning them usually requires inserting an exclamation mark somewhere in the previous command, which sometimes isn't possible. That's definitely not optimized for the common case. `confirm` instead prompts you, making it exactly one keystroke for each choice, and with saving being conveniently available just by pressing `Enter`.

19 Loading Files

- easy to load a file whose name is under the cursor
  - `gf` and `Ctrl+W f`
- useful for examining files from a list

```
% find -name '*.css' | gvim -
```

  - also with `grep`, `diff -r`, `cvs up`

`Ctrl+W f` is handy with `#include`-type statements, for opening a related file in another window.

20 Repetitive Shell Commands

- shell loops can be awkward and error-prone
- using *Vim* instead:
  1. generate the list:

     ```
     % ls *.gif | vim -
     ```

  2. transform into commands:

     ```
     :g/TEMP/d
     :%s/\v(.*)\.gif/convert & \1.png
     ```

  3. run them:

     ```
     :w ! sh
     :q!
     ```

When you've got a large number of files that need processing in some way, entering the commands by hand is far too tedious. Often using a shell `for` loop isn't much better. First there's the syntax to remember: not just the actual loop construct, but the variable expansion like `${f#gif}` for extracting bits from filenames.

Also of concern is the one-shot approach: you write all of your loop, then run it and see what happens. For anything even slightly complicated it can seem quite risky to chance that all the syntax is correct.

It isn't particularly obvious, but *Vim* can be used as a loop. First pipe the list of filenames, preferably one per line, into *Vim*. Then use as many commands as needed to transform each line into the Unix command to run on that file, and pipe the list into a hell.

This is safer because you get to preview exactly which commands will be run. It's often easier, because you can use multiple steps to transform the filenames into commands, and if you get a step wrong you can simply undo it.

Presenter Notes

- guide photos on website — shrinking, and renaming

21 Complex Transformations

- repetition
  - boring
  - surprisingly frequent
- worth learning a few *Ex* commands
- substitution:

  ```
  :%s/£/&pound;/g
  ```

- select lines to operate on:

  ```
  :g/\v^\=+$
  :g/\v^\=+$/ -p
  :g/\v^\=+$/ -s#.*#<h3>&</h3>
  :g/\v^\=+$/ d
  ```

Old-style *Ex* (“colon”) commands are not easy to learn even by *Vim* standards, but if you bother to get familiar with them they are useful much more often than you’d think.

The last two lines in the `:g//` example above puts `<h3>` HTML tags around lines that are ‘underlined’ with a row of equals signs, then removes the equals signs. The first two `:g//` commands are merely exploratory `:print` statements, checking that the lines selected by the regexp are the appropriate ones.

Many sorts of transformations are possible in this way. If the bits between the subheadings are bullet lists denototed with asterisks, then they could be HTMLified with somegthing like the following.

- Start a bullet list after each heading:

  ```
  :g#\v/h3# s/$/^M<ul>
  ```

  (The `^M` is typed by pressing `Ctrl+V` then `Enter`.)

- End a bullet list just before each heading:

  ```
  :g/\v\<h3/ s#^#</ul>^M
  ```

  The first and last items will need correcting by hand.

- Mark-up each item:

  ```
  :%s#\v^ *\*(.*)(\n *[^ *].*)*#<li>&</li>
  :%s#\v^(<li>) *\*#\1
  ```


22 Summary

- way too much to learn quickly
- a few commands can have big effects
- a little customization greatly increases user-friendliness
- questions:
  - `comp.editors` usenet group
  - http://www.vim.org/community.php for mailing lists
- plug:
  - *Vim* for Perl talk at YAPC Europe, Belfast, September

Presenter Notes

- plenty of other things that could've mentioned but missed out