Mastering Vim

Damian Conway

Summary

(damian@conway.org)

Tip 1: Get vim

- If you're still using vanilla vi it's time to upgrade to vim
- Download it from: http://www.vim.org/download.php
- Binary distributions available for "troublesome" platforms

Tip 2: Seek help

- Type:

  :help
- Or for a specific topic:

  :help <*topic*><CR>
- Or for a list of topics containing a given string:

  :help <*string*><TAB>
- Or for any topic matching a given (vim-ish) pattern:

  :helpgrep <*pattern*>
- Then :cnext<CR> to step through matches
- Within help files, anything in vertical bars is a hyperlink
- When the cursor is within bars, hit CTRL-] to jump to that topic
- At any time, hit CTRL-T to back out of the sequence of links

Tip 3: Understand the metaphors

- Vi-like editors all work on the same basic set of principles
- Understanding them helps make sense of the enormous complexity of the vim interface

Modes

- Almost all versions of vi are *modal*
- Usually in one of two basic modes...
- *Normal mode*, where you can move around and make large large-scale changes to the text
- *Insert mode*, where you can type in literal text

- Other modes include:
- *Command-line*, where you're typing an extended command after a colon
- *Visual* and *Visual-Block*, where you're selecting text prior to manipulating it
- *Replace*, where you're typing literal text over the top of existing text
- Experienced `vi` users instinctively know what mode they're in
- For novices, it's handy to set the `:showmode` option:

  ```
  :set showmode<CR>
  ```
- If you become "modally confused" just hit `<ESC>` to go back to Normal mode

Buffers

- `Vim` stores each text you edit in a *buffer*
- Typically each buffer is associated with a file and filename
- But that's not necessary: you can use nameless buffers too
- Changes to the text are updated in the buffer immediately
- But changes are only propagated back to the file when you `:write` the buffer (more on that later)
- If `vim` is terminated unexpectedly, you can generally recover the buffers you were working on (more on that later too)

Commands

- `Vim` is command-driven
- Typically from Normal mode
- Though there are commands in other modes too
- Three basic types of commands:
- Normal-mode commands (take a count before)
- Normal-mode operators (take a count before and an argument after)
- Command-line mode commands (introduced by a colon, take a range before and arguments after)

Tip 4: Learn your alphabet

- `Vim`'s normal-mode commands are the most frequently used
- Most users use only a small subset
- As an exercise, go through the keyboard...

(damian@conway.org)

- Almost every key (and shifted key, and control key) does something in `vim`
- How many do you know?
- How many do you use regularly?
- Increase your repertoire

Tip 5: Quit smarter

- The usual way people quit (from Normal mode) is:

```
:q
No write since last change (add ! to override)
:w
:q
```

- There's an abbreviation:

```
:wq
```

- There's an abbreviated abbreviation:

```
:x
```

- Or directly from Normal mode:

```
ZZ
```

Tip 6: Learn to move

- Most `vim` users know the basic cursor motions of Normal mode:

```
   k
 h   l
   j
```

- There are also higher-level motion commands and operators
- For words, lines, and other "text objects"

Word motions

- To move forward to the start of the next word: `w`
- To move backwards to the start of the previous word: `b`
- To move forward to the *end* of the next word: `e`
- To move backwards to the *end* of the previous word: `ge`
- "Words" are considered anything that is delimited by non-identifier characters

- All the motions have uppercase versions that use whitespace as the word delimiter instead

Line and paragraph motions

- To move to the start of the current line: `0` (zero character)
- To move to the start of the first word of the current line: `^`
- To move to the end of the current line: `$`
- To move to the start of the next line: `<CR>`
- To move to the start of the previous line: `-` (minus sign)
- To move to the start of the current paragraph: `{`
- To move to the end of the current paragraph: `}`
- A paragraph is delimited by empty lines (*not* blank lines)

Miscellaneous motions

- To move to the top of the buffer: `gg`
- To move to the end of the buffer: `G`
- To move to line *n*: *n*`G` or *n*`gg`
- To move to the point *p* percent through the buffer: *p*`%`
- Tip: if you:

```
:set showcmd
```

- ...you are shown the partial commands as you type them

Matching delimiters

- To move the matching bracket of a {...}, (...), or [...] pair: `%`
- By default, `vim` only matches {...}, (...), and [...]
- But you can extend that to whatever pairs you like:

```
set matchpairs+=<:>,«:»
```

- Even to "pairs" that aren't normally considered pairs:

```
set matchpairs+==:;
```

Repeated motions

- Tedious to have to type `11111` to move five chars left
- Instead can type: `51`

(damian@conway.org)

- Likewise, up three lines: `3k`
- Likewise, ahead four paragraphs: `4}`

Scrolling a large buffer

- To scroll *F*orward one window's worth of text: `<CTRL-F>`
- To scroll *D*own a half window's worth of text: `<CTRL-D>`
- To scroll *U*p one half window's worth of text: `<CTRL-U>`
- To scroll *B*ack a full window's worth of text: `<CTRL-B>`

Tip 7: Keep track of where you are

- Type `<CTRL-G>` to be told where you are
- Type `g<CTRL-G>` to be told more precisely where you are
- Better still, enable the ruler:

```
:set ruler
```

- Shows: *<line>,<column> <percentage>*
- But the information the ruler shows is highly configurable
- See:

```
:help rulerformat
:help statusline
```

Tip 8: Keep track of where you were

- In Normal mode, you can leave a *mark* at any cursor position
- Just type `m<char>`
- For example:

```
mh
```

- Then to go back to that mark, you type a backtick: `` `<same char> ``
- That is:

```
`h
```

- Or, to go to the start of the line containing the mark, type a single quote: `'<same char>`

```
'h
```

- If you use a lowercase letter, the mark is per-buffer

- If you use an uppercase letter, the mark is global
- For example:

```
mQ
:edit some_other_file<CR>
`Q
```

- The `` `Q `` would immediately switch your buffer to the file in which the mark was set, then jump to the mark
- You can see all the marks you've set by typing: `:marks`

Editing with marks

- One of the most important uses of marks
- As a "motion" for other editing commands
- For any command that takes a motion after it...
- ...(such as `c`, `d`, `y`, `>`, etc)...
- ...you can use a jump-to-mark motion as well
- For example, to delete several hundred lines of code...
- ...move to the start of the code...
- ...set a "start" mark: `ms`
- ...move to the end of the code...
- ...delete back to the "start" mark: `` d`s ``

Tip 9: Use `vim`'s own navigation marks

- `Vim` sets certain non-alphabetic marks automatically as you work
- The most valuable is double backtick, the *context mark*: `` `` ``
- `` `` `` takes you to the last place you jumped from
  (where "jumped" means searched, used `G`, or returned to a mark)
- A handy trick when working on two regions of a file...
- ...set the context mark at the first region: `` m` ``
- ...jump to the second region...
- ...then `` `` `` between the two regions
- Another handy "Where was I?" mark is backtick-doublequote: `` `" ``
- This is where you were last time you exited the current file
- A useful shell alias:

(damian@conway.org)

```
> alias v \!vim +normal\\\'\\\"
```

- `vim` also keeps a list of all your jumps within a file
- You can use `<CTRL-O>` and `<CTRL-I>` to step back and forth through the list

Tip 10: Learn how to change text efficiently

- You can get into Insert mode via:
- `i` : Start inserting before the current cursor position
- `I` : Start inserting before the start of the current line
- `a` : Start inserting after the current cursor position
- `A` : Start inserting after the the end of the current line
- `o` : Start inserting on a new line below the current line
- `O` : Start inserting on a new line above the current line
- `s` : Delete the current character and start inserting from there
- *N*`s` : Delete the next *N* characters and start inserting
- You can get into Replace mode via:
- `r` : Replace the character under the cursor then return to Normal mode
- `R` : Replace characters from the current cursor
- `c`*<motion>* : Replace from the current cursor position to where the motion reaches
- `C` : Replace the current line

Tip 11: Repeat yourself numerically

- If you prefix any of the preceding commands with a number, it is repeated that many times
- For example:

```
30a_-<ESC>
```

- ...produces:

```
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-
```

- Likewise:

```
5oPlease!<ESC>
```

- ...produces:

```
Please!
Please!
Please!
Please!
Please!
```

- You can also repeat the most recent editing command quickly with:

  `.`

- For example, the earlier begging could also be achieved with:

  `oPlease!<ESC>`

  `.....`

- Has the advantage that you can simply stop when you have enough...
- ...rather than having to guesstimate how many beforehand....

Tip 12: Fix your deletions

- Typing a backspace/delete in either editing mode deletes the character preceding the cursor
- Normally you can only backspace back to the point where you started inserting/replacing
- However, `vim` lets you set an option that extends the reach of deletion
- For example:

```
:set backspace=start    " Can delete back past start of edit
:set backspace=indent   " Can delete back past autoindenting
:set backspace=eol      " Can delete back to previous line
```

- Typically you want all three:

```
:set backspace=indent,eol,start
```

Tip 13: Control your insertions

- In either editing mode *most* characters you type insert that character...but not all
- Many of the control characters have special insertion behaviours
- `CTRL-Y` duplicates the character in the same column on the preceding line
- `CTRL-E` duplicates the character in the same column on the following line
- `CTRL-A` inserts again whatever the most-recent inserted text was
- `CTRL-R` inserts the contents of a register (more later)

(damian@conway.org)

- CTRL-R= evaluates an expression and inserts the result
- CTRL-T inserts a tab at the start of the line (without moving the insertion point)
- CTRL-V inserts the next character verbatim (even if it's normally a control character)
- CTRL-W deletes the word preceding the cursor
- CTRL-O takes you back to Normal mode for one command
- Handy, for example, to clean the rest of the line: ^OD

Tip 14: Search smarter

- To search for instances of a regex within a buffer:

  /<pattern><CR>
- Takes you to the next piece of text after the cursor that matches the pattern
- To go the second match, either

  /<CR>
- Or (more commonly):

  n
- To search backwards:

  ?<pattern><CR>
- And backwards again:

  N
- Can configure vim to wrap the search around from the end of the buffer back to the start:

  :set wrapscan
- Regexes used are sed-ish, but vim extends them considerably
- Comparable in power to Perl, but numerous differences in syntax
- Appendix A summarizes the syntax

Searching within a line

- Often you just want to jump to a particular character within the current line
- There's a short-cut for that: f<char> (for "find")
- For example:

  fu
- You can jump backwards too: F<char>

- If you put a count before the `f` you're jumped to the Nth instance:

```
5fu
```

Searching for existing word

- If you have the word you want to search for already under your cursor...
- ...just type `*` to find the next instance
- ...or `#` to find the previous

Case (in)sensitivity

- Normally `vim` searches are case-sensitive
- If you would prefer case-insensitive set the following option:

```
:set ignorecase
```

- Better still, if you'd like "partial sensitivity", set this option as well:

```
:set smartcase
```

- "Smartcase" overrides the "ignorecase" behaviour whenever your pattern includes any uppercase letters
- Even with these options set you can still be specific when you want
- Use the `\c` specifier within the pattern
- Everything after it within the pattern will be match case-insensitively, no matter what
- Likewise, there's `\C` to unconditionally turn case sensitivity on

Tip 15: Use searches as motions

- Because every successful search results in a movement of the cursor...
- ...every search command can be used as the *<motion>* for any other operator
- For example, to yank every line up to the `__END__` line of a file:

```
y/__END__<CR>
```

- Or to delete everything within a line, from the cursor up to and including the semicolon; type this:

```
df;
```

- Often, of course, you'd rather delete everything up to but *excluding* the semicolon (or whatever)
- So `vim` also has the `t<char>` (or "to") command
- By itself, jumps to the character *before* the specified character

(damian@conway.org)

- As a motion, specifies "up to but excluding":

```
dt;
```

Tip 16: Preview your search results

- `Vim` has an option that causes it to show you where your search will match:

```
:set incsearch
```

- Looks ahead as you type the search pattern, and highlights first match
- Only jumps to that position when you hit `<ENTER>`
- If you're looking ahead and hit `<ESC>` instead...
- ...cancels the search and returns cursor to former position
- Very handy to "peek-and-return"

Tip 17: Highlight your search results

- Often the match you find isn't the one you want
- So it can be handy to be shown where else in the buffer you should look
- Search highlighting makes that easy:

```
:set hlsearch
```

- Now every match of every search will be highlighted
- Search still jumps to the first match
- But now you can see where else you might need to look
- However, the highlighting persists until your next search
- Annoying
- To get rid of it, type:

```
:nohlsearch
```

Tip 18: Search and destroy

- You can also tell `vim` to find a match and replace it:

```
:s/<pattern>/<replacement>
```

- With find the next match and replace that one instance
- You can specify a range of lines to restrict the search to:

```
:10,33s/<pattern>/<replacement>
```

- Will then replace first instance per line in that range
- If the range consists of a single number, only that line is modified
- For example, only change the first line:

```
:1s/<pattern>/<replacement>
```

- If you include a sign, then the range will be relative to the current line:

```
:-10,+33s/<pattern>/<replacement>
```

- That means: "from 10 lines above the cursor to 33 lines below it"
- If you use a semi-colon instead of a comma:

```
:-10;+33s/<pattern>/<replacement>
```

- ...then the end of the range is relative to the start of the range, instead of relative to the current line
- So the semicoloned version means: "from 10 lines above the cursor for the following 33 lines"
- In a range, `.` means the current line, `$` means the last line
- For example, to substitute on each line in the rest of the file:

```
:.,$s/<pattern>/<replacement>
```

- There's also a short-cut for making the range "the next `N` lines"
- If you enter a number before the colon:

```
99:
```

- You get:

```
:.,+98
```

- You can specify the entire file as the range:

```
:1,$s/<pattern>/<replacement>
```

- And there's a short-cut:

```
:%s/<pattern>/<replacement>
```

- You can also specify a range of lines according to their contents
- Use `/<pat>/` to specify "the next line that matches *<pat>*"
- Use `?<pat>?` to specify "the previous line that matches *<pat>*"
- For example, to substitute only within the body of an HTML file:

```
1G
:/<body>/,/<\/body>/s/<I>/<EM>/
```

- You can add an offset after either range specifier
- For example, to match from the line after the previous instance of "foo"...
- ...to ten lines before the end of the file:

```
:?foo?+1,$-10s/<pattern>/<replacement>
```


(damian@conway.org)

- Note that many other "colon" commands take ranges
- The specification syntax is always the same

Multiple substitutions

- A substitution only substitutes one match per line
- Even if you use the `%` range, you get one substitution on every line
- To specify a substitution of *every* match on a line...
- ...append the `/g` modifier:

```
:s/<pattern>/<replacement>/g
:10,33s/<pattern>/<replacement>/g
:%s/<pattern>/<replacement>/g
```

Cautious substitution

- To request a confirmation on each replacement append a flag: `/c`

```
:%s/cat/feline/c
```

Substitute again

- Often you want to repeat the last substitution on another line
- You can do that with just:

```
:s<CR>
```

- Or use the Normal-mode shortcut:

```
&
```

- You can repeat the substitution globally with:

```
:%s
```

- Or:

```
g&
```

- Handy to "get it right" on one line, then apply everywhere

Tip 19: Copy shamelessly

- To copy some text, you "yank" it, with the `y` operator
- To copy a line: `yy` or `Y`
- To copy a word: `yw`
- To copy a paragraph: `y}`

Pasting text

- Once you've copied or cut some text, it can be pasted elsewhere using `p`
- For example, to move a paragraph containing the word "rabbit" to the end of the buffer:

```
/rabbit<CR>
dap
G
p
```

- `p` always pastes *after* the cursor
- ...like an `a` if you copied/cut less than a full line
- ...like an `o` if you copied/cut a full line or more
- You can also paste *before* the cursor: with `P`

Indented pasting

- To paste lines above or below the current line...
- ...but with the same level of indenting:

```
]p
]P
```

Targeted copy-and-paste

- If can move a chunk of text to a specific location in one command:

```
:<range>copy <target>
```

- The *<target>* uses the same specification syntax as ranges
- For example, to copy the current line to the end of the file

```
:copy $
```

- For example, to copy the current paragraph to after the first "__END__" marker:

```
:?^$?,/^$/copy /__END__/
```

Targeted cut-and-paste

- Instead of copying text to a target, you can move it:

```
:<range>move <target>
```

- For example, to move the next 10 lines to the start of the file:

```
:+1,+10move 0
```

Tip 20: Store text in registers

- Every `y` or `d` command stores the copied text in a special location
- Known as a *register*
- The default register is nameless
- Other registers are named with lowercase letters: `"a` `"b` `"c` etc.
- To yank text into a named register, specify the register name before the yank command:

  ```
  "ayw
  ```

- Likewise to delete, but save the deleted text in a named register:

  ```
  "zd$
  ```

- To paste the contents of a register, name the register before the paste command:

  ```
  "ap
  ```

- In Insert mode, to "type" the contents of register `"n`

  ```
  <CTRL-R>n
  ```

- In Insert mode, to "insert" the contents of register `"n`

  ```
  <CTRL-R><CTRL-R>n
  ```

- Registers are useful for keeping snippets that you need to use over and over
- If you name the register using an uppercase letter, it's the same as the lowercase version...
- ...except that the yanked/deleted text is *appended* to the register instead of overwriting it
- Useful for trawling a document and picking out the bits you want

Tip 21: Travel through time

- `Vim` has the ability to undo arbitrarily many changes to a buffer
- And to redo them if you decide they were okay after all
- To undo the last buffer change: `u`
- Note that, unlike `vi`, `vim`'s undo doesn't undo a preceding undo
- To redo the last undone change(s): `<CTRL-R>`

(damian@conway.org)

Branched undo

- Prior to version 7, if you undid some changes then made a new change...
- ...`vim` threw away all the undone changes
- Since version 7, instead of discarding this "alternate history"...
- ...`vim` branches out into a new history
- ...but remembers the old one
- The u and <CTRL-R> commands run you back-and-forth within the most recent historical branch
- But every change in every branch is timestamped
- So you can go back/forward to any particular point in your editing history
- Move back in time (through different branches) with: `g-`
- Move forward (through different branches) with: `g+`
- Move back/forward to a particular time with:

```
:earlier <time_offset>
:later   <time_offset>
```

- For example, to return to the state the buffer was in 10 minutes ago:

```
:earlier 10m
```

- Then to move forward again to the state 30 seconds after that time:

```
:later 30s
```

Tip 22: Read and write files efficiently

- To edit a file from within `vim`:

```
:edit <filename>
:next <filename>
```

- The difference is that `:next` autowrites if the `autowrite` option is set
- To edit the next/previous file listed on the command line:

```
:next
:prev
```

- To edit the file you just left:

```
:next #
```

- To edit the file whose name/path your cursor is over:

```
gf
```

Reading buffers

- To read the contents of another file into the buffer, inserting them below the current line:

```
:read <other_filename>
```

- To read in the output of another process, inserting it below the current line:

```
:read !<process>
```

- For example:

```
:read !GET http://www.mycorp.com/std_disclaimer.html
```

- If you specify a line-number before `read`, the new text is inserted after that line
- The commonest use is to insert something at the very start or end of a buffer:

```
:0read std_header
:Gread std_footer
```

Tip 23: Use a filter

- Can also write buffer to a process then read back in from that process

```
" Sort entire buffer (range is %)...
:%!sort
```

- Can specify other ranges of lines:

```
" Sort current line and next 20 lines...
:.,.+20!sort
```

- Or filter a single line:

```
!!wc
```

- Or for a motion:

```
" Sort from here to EOF...
!Gsort

" Sort surrounding paragraph...
!ipsort
```


(damian@conway.org)

Tip 24: Sort internally

- But what if your system doesn't provide a `sort` utility?
- `Vim` has its own built-in sorting mechanism:

```
:sort
```

- And reverse sort:

```
:sort!
```

- Many convenient options:

```
:sort n             Sort numerically (by first number in line)
:sort x             Sort numerically (by first hexadecimal number)
:sort o             Sort numerically (by first octal number)
:sort i             Sort case-insensitively
:sort u             Remove duplicate lines after sorting
:sort /pattern/     Sort after skipping text matched by pattern
:sort r /pattern/   Sort using text matched by pattern
```

- The options can be combined:

```
:sort iur /^.\{6}/  Sort case-insensitive, unique, on 1st 6 chars
```

Tip 25: Remember your history

- Whenever you execute a "colon" command or a search, `vim` remembers it
- Next time you type a colon or a slash, you can trawl through that history
- Just hit the `<UP>` or `<DOWN>` key
- When you find the one you want, just hit return
- If you specify part of the pattern or command, the arrows only show patterns/commands with the same prefix
- Alternatively, you can *edit* the history to create a new command or search
- Instead of `:`, type `q:` to initiate the command
- Instead of `/`, type `q/` to initiate the search
- Navigate to the command/pattern you want
- Modify it using the usual text-editing commands
- When you hit `<RETURN>` the modified line will be executed

Tip 26: Autocomplete your text

- Once of the most useful features of modern shells
- No longer have to type in full filenames
- Just the first few letters...
- ...then hit <TAB>
- Typically shown the list of possibilities
- If you're lucky, subsequent <TAB>s cycle each in turn
- Likewise for command names at the start of the line

Filename completion

- Can do the same in `vim`...
- ...when entering a command that expects a filename
- `:edit` or `:write`, for example
- Type the first few letters and hit <TAB> to cycle the possibilities
- Alternatively, type <CTRL-A> to insert all possibilities

Navigating a completion

- Once completion is active, <TAB> cycles to the next possibility
- So does <CTRL-N>
- To go back to a previous possibility: <CTRL-P>

Smarter completion

- You can configure `vim` to be even more helpful
- Depending on how you prefer your completions
- Using the `wildmode` command:

```
set wildmode=list
```

- Now completions just list the possibilities (but never fill them in)
- Or:

```
set wildmode=list:longest
```

- Now completions list the possibilities and fill in the longest common prefix
- Or:

```
set wildmode=list:longest,full
```


(damian@conway.org)

- Now completions list the possibilities and fill in the longest common prefix
- And subsequent <TAB>'s at the same place cycle the full possibilities
- See `:help wildmode` for the numerous other possibilities
- You can also change the character that initiates completion:

```
:set wildchar=<ESC>
```

Selective completion

- You probably never want to complete to a `.o` or `.obj` file
- So you can tell `vim` that:

```
set wildignore+=*.o
set wildignore+=*.obj
set wildignore+=core
```

Other forms of completion

- Completion is so useful that `vim` supports it just about everywhere
- When typing a "colon" command, <TAB> completes valid `ex` commands
- When typing in a shell command (after a `:!`), <TAB> completes valid shell commands
- After a `:set`, <TAB> completes all valid `vim` option names
- Likewise after a `:map`, <TAB> completes existing macro names

Tip 27: Complete your text too

- Once you're hooked on completion, man, you need it everywhere
- Including in the text you're editing
- So `vim` provides the <CTRL-X> mode
- While in Insert or Replace mode, type <CTRL-X> after a partial "word"
- Then select the type of completion you want by typing a second control key...

Filename/filepath completion

- <CTRL-X><CTRL-F> completes filenames and filepaths

Defined symbol completion

- <CTRL-X><CTRL-D> completes predefined C preprocessor symbols

```
#define def_FUNC_CALL f(o*o)
result = my de<CTRL-X><CTRL-D>
```

- Can also change what it thinks a definition is
- Set the `define` option to a pattern
- For example, to complete defined Perl subroutines:

```
:set define=^\\s*sub
```

- Then:

```
sub defenestrate_exception {...}
$result = de<CTRL-X><CTRL-D>
```

Identifier completion

- <CTRL-X><CTRL-N> completes identifiers from the current file
- Identifiers are sequences of "keyword" characters
- The default keyword characters are: '@','0'-'9','_','a'-'z','A'-'Z'
- Can change this default by setting the `iskeyword` option
- For example, to handle C identifiers:

```
:set iskeyword=a-z,A-Z,48-57,_,.,-,>
```

- Can also supply a list of "standard" identifiers that should always be considered for completion
- Put the list in a separate file
- Then tell the completion mechanism to use it as well:

```
set complete+=k~/data/my_std_identifiers
```

- The leading `'k'` indicates that the file is laid-out like dictionary file
- That is, one word per line
- Can also search for identifiers from `#include` files
- Use <CTRL-X><CTRL-I>

Other completions

- <CTRL-X><CTRL-L> completes existing lines
- <CTRL-X><CTRL-K> completes identifiers from a dictionary file
- <CTRL-X><CTRL-T> completes *related* identifiers from a thesaurus file

(damian@conway.org)

- See:

```
:help ins-completion
```

Tip 28: `Vim` is a file browser

- `Vim` comes with a standard plug-in that helps navigate directory structures
- If you edit a directory, `vim` acts like a simple file browser
- Hit `<ENTER>` to descend to a particular file or subdirectory
- Hit `-` to ascend to the parent directory
- Hit `D` to delete a file
- Hit `R` to rename a file
- Hit `s` to switch between sorting by name, time, and size
- Hit `r` to reverse the sorting order
- Hit `i` to change how much information you're shown (and how compactly)
- Hit `p` to preview a file in a separate window
- Hit `c` to change the current directory to the directory under the cursor
- Hit `d` to create a new directory
- Hit `o` to open the selected dir/file in a horizontally split window
- Hit `v` to open the selected dir/file in a vertically split window

Tip 29: Explore `vim`'s many options

- Most of `vim`'s interface and behaviour is configurable
- Mostly by setting specific configuration options
- Can set them ad hoc during an editing session
- Use the `:set` command
- More effective to set them "permanently"
- Add them to your `.vimrc` file (in your home directory)
- To turn a boolean option on:

```
:set <option_name>
```

- To turn it off:

```
:set no<option_name>
```

- To toggle it:

```
:set inv<option_name>
:set <option_name>!
```

- To reset it to default:

```
:set <option_name>&
```

- To turn on an option that takes a value

```
:set <option_name>=<option_value>
```

- To add to/remove from a list of option values:

```
:set <option_name>+=<new_value>
:set <option_name>-=<existing_value>
```

- To see all current option values:

```
:set
```

Tip 30: Show your line numbers

- Some programmers like to see line numbers all the time:

```
:set number
```

Tip 31: Constrain your line widths

- Most people prefer their editor to wrap text lines that get too long
- Typically at 72, 78, 79, 80, or 132 columns
- To tell `vim` to do that as you insert:

```
set textwidth=78
```

- Alternatively, you can tell `vim` to wrap a certain number of characters before the edge of the terminal:

```
set wrapmargin=2
```

- Recommend `textwidth` rather than `wrapmargin`
- ...because `vim` won't automatically rewrap existing lines when the terminal width changes

(damian@conway.org)

Tip 32: Write before you leave

- By default, before leaving a buffer, `vim` asks you whether you want to save the contents
- You almost always do
- So make it the default:

```
:set autowrite
```

Tip 33: Configure your `.vimrc` file

- Once you've set your options the way you prefer them, exit you editor...and they'll all be lost!
- Setting individual options every time you edit is impractical
- So you can save the settings you prefer and have them automatically loaded each time you edit
- Such configuration settings are stored in a file named `.vimrc`
- Or `_vimrc` on certain filesystems
- Your personal `.vimrc` lives in your home directory: `~/.vimrc`
- It's a plain text file that you can create and edit (using `vim`!)
- Or, better yet, start up a `vim` session from your home directory...
- ...set your preferred options and modes...
- ...then type:

```
:mkvimrc
```

- Your options will be saved in a new `.vimrc` file in your home directory
- You can then add new configuration features by editing the file directly

Tip 34: Set your options by browsing

- `vim` has so many options that setting them can be daunting
- In fact, just `remembering` them can be a challenge
- So `vim` provides an *option browser* to help
- Start it up by typing:

```
:options | resize
```

- Navigate using the normal motion commands
- Hit `<ENTER>` on a section name to jump to that section's options

- Hit <ENTER> on a description line for the full help entry
- (Type ZZ to get close the help window)
- Hit <ENTER> on a boolean option to toggle it
- For options that take values, edit the value and then hit <ENTER>
- Once you're happy with the options you've set, you can save them as before:

```
:cd ~
:mkvimrc
```

Tip 35: Defang those tabs

- Tabs create more editing problems than any other character
- They're ill-defined
- They change appearance under different defaults
- They're best avoided
- Or, at least, restrained

Sticking with eight-column tabs

- If you're forced to use eight-column tabs...
- ...you can still have your cake and eat their tabspacing too
- Vim provides a "soft tabs" option:

```
:set softtabstop=4
```

- When this option is set, vim uses the standard 8-column tabs
- But doesn't insert a tab when you hit <TAB>
- Instead, it inserts 4 spaces on the first <TAB>
- ...and then removes them and inserts a tab on the second <TAB>
- So it *feels* like you're using 4-column tabs
- ...and *looks* like you're using 4-column tabs
- ...but your code still looks right under 8-column tabspacing

Adjusting tabs

- If you want to adjust the tabs in a file from one tabspacing to another:

```
:set tabstop=<current tabspacing>
:retab <new tabspacing>
```


(damian@conway.org)

- For example:

```
:set tabstop=8
:retab 4
```

- If you go from a smaller to a larger tabspacing, some tabs will be replaced with sequences of spaces
- Which means that `:retab` does not "round-trip"
- For example, if your `tabstop` is originally 4, then:

```
:set tabstop=4
:retab 8
:retab 4
```

- ...will leave what were originally single tabs as 4-space sequences
- Use:

```
:retab! 4
```

- ...to tell `vim` to convert multiple-space sequences to tabs where feasible

Replacing tabs as you type

- If you set the `expandtab` option:

```
:set expandtab
```

- ...`vim` replaces every tab you type with the appropriate number of spaces
- ...as determined by the current value of `tabstop`
- If `expandtab` is set when a `:retab` is performed, all tabs are expanded to spaces:

```
:set expandtab
:retab
```

Tip 36: Indent cleverly

- If you decide to have tabs defanged using the `expandtab` option you create a different problem
- The <TAB> key and shift commands (<< and >>)
- Under `expandtab` they insert a fixed number of spaces
- So things stop lining up
- You can overcome that, by telling `vim` to "round down" any tabspacing:

```
set shiftround
```

- Then tabs and shifts always tab and shift to the next tabstop

(damian@conway.org)

Smarter indenting

- Normally, if you're expanding tabs, a tab inserts one `tabstop` worth of space everywhere
- That's fine, except at the start of a line
- Because the shift commands insert one `shiftwidth` worth of space instead
- So your indenting can get messed up if you set `shiftwidth` and `tabstop` to different values
- Don't do that!
- But, if you *must*, you can tell `vim` to use `shiftwidth` for column-1 tabs:

```
:set smarttab
```

Total tabular control

- If you frequently have to work with "alien" tabspacings a simple set of macros can make life very much easier:

```
map <silent> TR :set   expandtab<CR>:%retab!<CR>
map <silent> TT :set noexpandtab<CR>:%retab!<CR>
function ConvertToTabSpacing (newtabsize)
    let was_expanded = &expandtab
    normal TT
    execute "set tabstop="     . a:newtabsize
    execute "set shiftwidth=" . a:newtabsize
    if was_expanded
        normal TR
    endif
endfunction


map <silent> T@ :call ConvertToTabSpacing(2)<CR>
map <silent> T# :call ConvertToTabSpacing(3)<CR>
map <silent> T$ :call ConvertToTabSpacing(4)<CR>
map <silent> T% :call ConvertToTabSpacing(5)<CR>
map <silent> T^ :call ConvertToTabSpacing(6)<CR>
map <silent> T& :call ConvertToTabSpacing(7)<CR>
map <silent> T* :call ConvertToTabSpacing(8)<CR>
map <silent> T( :call ConvertToTabSpacing(9)<CR>
```


(damian@conway.org)

- `TT` inserts tabs everywhere possible
- `TR` removes every tab (via `expandtab`)
- The `ConvertToTabSpacing` function ensures tabs are in effect...
- ...then sets new values for `tabstop` and `shiftwidth`
- ...then re-expands to spaces, if you were originally expanded
- The mappings `T@` through `T(` then provide quick conversion to the corresponding tabspacings

Tip 37: Recover after a disaster

- Inevitably your `vim` session will eventually crash
- Not because `vim` is unreliable (it's incredibly reliable)
- But because your hardware, or operating system, or filesystem, or network fails
- As you edit, `vim` periodically swaps out the current state of your buffer
- Writes it to a file in the same directory
- If your `vim` session is prematurely terminated, can recover most of your work from that file

Recovering after a crash

- For example, if you were editing:

```
> vim my_vital_data.txt
```

- And the session was unexpectedly terminated, try:

```
> vim -r my_vital_data.txt
```

- If there is a swap file available, `vim` will use it to reconstruct your buffer
- It's typically a good idea to immediately save that buffer under a related name:

```
:w my_vital_data.recovered
```

- ...just in case things fall over again
- You can find all the existing swap files using:

```
> vim -r
```

- (i.e. with no filename)

Preventing mishaps

- `Vim` also uses the swap files to detect when you attempt to edit a file that another `vim` session is already editing

- If you do that, the second vim process will inform you:

```
E325: ATTENTION
Found a swap file by the name "my_vital_data.swp"
        owned by: damian   dated: Mon May 22 15:59:22 2006
        file name: ~damian/Talks/Vim/my_vital_data
        modified: yes
        user name: damian   host name: Gort.local
        process ID: 2849 (still running)
While opening file "my_vital_data"
             dated: Mon May 22 15:59:07 2006
(1) Another program may be editing the same file.
    If this is the case, be careful not to end up with two
    different instances of the same file when making changes.
    Quit, or continue with caution.
(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r my_vital_data"
    to recover the changes (see ":help recovery").
    If you did this already, delete swap file "my_vital_data.swp"
    to avoid this message.
Swap file "my_vital_data.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort:
```

- "Open Read-Only" or "Abort" is almost always the correct response here

Improving your chances

- Normally vim writes out your buffer to a swap file every 200 keystrokes
- And doesn't always sync that swapfile to disk when it's written (that depends on your O/S and filesystem)
- So you could potentially lose 400 (or more!) keystrokes in a crash
- As long as you don't mind the near-constant disk activity, your session will be much more recoverable with:

```
:set updatecount=10
:set swapsync
```

- Then you'll never lose more than about 20 keystrokes
- However both these may adversely affect your interactivity, as well as compromising battery life on laptops
- swapsync in particular

(damian@conway.org)

Tip 38: Make a backup

- Normally when you write a file:

```
:write
:write <filename>
```

- ...the buffer contents overwrite the corresponding disk file
- If you want a little more safety:

```
:set backup
```

- Now `vim` will make a backup of the file before overwriting it
- The backup has the original filename with a ~ or _ appended
- You can change that extension with (for example):

```
set backupext=.bak
```

- Normally, the backup is written to the same directory as the original file
- But you can nominate another directory
- For example:

```
set backupdir=~/.backups
```

- There are numerous other options besides these: when to skip backups, what to do with old backups, how to create the backup, etc.
- For more details, see:

```
:help backup
```

Tip 39: Edit visually

- One of the challenges of `vim` is remembering the many ways to specify what your command is suppose to act upon
- For example:

```
x     delete character
10x   delete 10 characters
diw   delete inner word
daw   delete a word
dd    delete one line
dtz   delete up to next letter 'z'
dfz   delete up to and including next letter 'z'
dib   delete inner '(' ')' block
dap   delete a paragraph
```

- If you're a visually oriented person, it's hell
- So `vim` provides *visual* modes as well
- In these visual modes the usual *command-target* sequence is reversed
- You first specify the area to be affected
- Then specify the effect

Visual mode

- The simplest visual mode is Visual mode
- You type `v` and you're in it
- Then you move around, using any of the normal motion commands
- As you move, `vim` highlights from where you started to where you are now
- Once you've selected the text you want, you type the command you want to execute
- You can execute any "colon" command
- Or any of the normal commands: `y`, `>`, `~`, etc.
- For example:

```
v$d
v2jd
v}d
```

- Note that line-oriented commands (colon commands, shifts, etc.) operate on the full lines covered by the selection

Visual-Line mode

- Because you often do want to operate on complete lines, there's a second visual mode
- Use `V` instead of `v`
- Selects entire lines as you move the cursor
- No matter where in the line the cursor is
- Otherwise identical to `v` mode

Visual-Block mode

- The funkiest and most useful visual mode
- Enter it using `<CTRL-V>`
- Selects a rectangular block from your cursor position to wherever you move

(damian@conway.org)

- Subsequent command is applied only to that block
- Incredibly handy for adjusting table columns
- Special commands for block mode

`<CTRL-V><motion>I<text>`	Inserts text before block on every line
`<CTRL-V><motion>A<text>`	Appends text after block on every line
`<CTRL-V><motion>c<text>`	Changes every line of block to text
`<CTRL-V><motion>r<char>`	Changes every character in block to char

Terminating block mode

- If you're in block mode and decide you'd rather not be just hit `<ESC>`

Visually selecting text objects

- Normally a visual selection extends from where you entered your visual mode...
- ...to wherever your cursor is now
- However, as well as using motion commands to specify your selection...
- ...you can also use "text object" commands
- For example, to indent the paragraph surrounding the cursor:

```
vip>
```

- Another handy variation joins all the lines of your current paragraph:

```
vipJ
```

Tip 40: Abbreviate your typing

- It's tiresome to have to type in oft-repeated sequences
- For example, your email or web address
- Or standard text markers like:

```
-----cut----------cut----------cut----------cut------
```

- Or even just repeated tags like:

```
<blockquote><cite>
    ...
</cite></blockquote>
```

- So `vim` provides an mechanism that allows you to specify abbreviations that will be expanded when typed

- For example:

```
:abbreviate hdco http://damian.conway.org
:abbreviate daco damian@conway.org
:abbreviate bqc <blockquote><cite><CR></cite></blockquote>
:abbreviate --c <CR>-----cut---------cut---------cut-----
```

- When the LHS of an abbreviation is recognized during insertion of on the command line it is immediately expanded to its RHS
- The expansion is exactly like typing it yourself
- So, as long as you're careful, you can be even cleverer:

```
:ab bqc <blockquote><cite><CR><CR></cite></blockquote><UP><TAB>
:ab --c <CR>----cut-----cut-----cut----<C-O>:center<CR><DOWN>
```

Restrictions on the LHS

- There are restrictions on what you can use as an abbreviation
- The LHS of an `:abbreviate` must either consist entirely of keyword characters (i.e. be an identifier)
- ...or it must consist entirely on non-keyword characters but end in a single keyword character (e.g. `#1` or `--c` or `@=x`)
- ...or it must end in a non-keyword character, with the preceding characters being anything you like (e.g. `1#` or `c--` or `orz!`)
- You can't use whitespace in abbreviations

Expanding abbreviations

- Abbreviations also require some trailing context to know they've been entered
- That is, you have to type a non-keyword character after the abbreviation before it will be expanded
- The extra character you typed is inserted *after* the expansion
- You can expand an abbreviation without trailing context by typing `CTRL-]`

Reviewing your abbreviations

- If you type:

```
:abbreviate
```

- ...without an argument, you get a list of the active abbreviations
- To remove an abbreviation:

```
:unabbreviate bqc
```


(damian@conway.org)

- To remove all abbreviations:

```
:abclear
```

Targeted abbreviations

- Be careful though
- Abbreviations are active on the command line too
- To deactivate them type a literal <CTRL-V> before the abbreviation
- (You have to type two <CTRL-V> to get one though)

```
:abbreviate ^V^V--c
```

- On the other hand, abbreviations *can* be useful on the command line
- For example, if you frequently need to save to a particular file:

```
:abbreviate bak /usr/local/tmp/backup/damian/checkpoint
```

- However, you'll also get that expanded in inserted text
- A better option is to use the `iabbrev` and `cabbrev` versions
- With these you can tell `vim` exactly where to expand the abbreviation
- Only in insertions:

```
:iabbrev bqc <blockquote><cite><CR><CR></cite></blockquote><UP><TAB>
```

- Or only on the command line:

```
:cabbrev bak /usr/local/tmp/backup/damian/checkpoint
```

Computed abbreviations

- You can specify that an abbreviation should expand to the result of some expression in the vimmish command language
- For example, to have `TS` expand to the current timestamp:

```
:abbreviate <expr> TS strftime("%c")
```

- Or to have `PPP` expand to the last yanked text:

```
:abbreviate <expr> PPP getreg('')
```

- Or to have `^^` insert the contents of the preceding non-empty line:

```
:abbreviate <expr> ^^  getline(search('\S\_.*\n\_.*\%#','b'))
```

Tip 41: Map your commands

- Abbreviations are great, but suffer from two major constraints: they're only available in Insert mode and on the command line, and they require an extra character typed after them
- Maps remedy both those problems
- A map is a character sequence that is expanded as soon as the complete sequence is typed (no trailing context required)
- You can specify separate maps for Insert mode, Command-line mode, Normal mode, Visual mode, and various combinations thereof

Insertion maps

- Insertion maps can be used instead of abbreviations:

```
:imap ww http://damian.conway.org
:imap ee damian@conway.org
:imap ;b <blockquote><cite><CR></cite></blockquote><ESC>O
```

- Maps have the advantage that they don't require an additional character to be typed
- They have the disadvantage that they don't require an additional character to be typed
- That means you need to be careful in selecting your map trigger
- In practice, abbreviations are better for *content* expansions:

```
:ab NAME Dr Damian Conway
:ab ADDR Thoughtstream Pty Ltd<CR>PO Box 668<CR>Ballarat
```

- ...whilst imaps are better for behavioural insertions
- For example, if you don't use tabs and would prefer <TAB> to do word completions:

```
:imap <TAB> <C-N>
```

- Now every tab character immediately acts like a <CTRL-N> completion request

Normal maps

- The second advantage of maps is that they can be applied in Normal mode
- For example, if you're forever deleting paragraphs:

```
dip
```

- ...you could abbreviate that command:

```
:nmap X dip
```


(damian@conway.org)

- Likewise, if you find you want Visual Block mode far more often than Visual mode...steal Visual mode's trigger:

```
:nmap v <C-V>
```

- Or to make `vim` more browser-like:

```
:nmap <Space> <PageDown>
```

- Or to make interfile navigation more convenient:

```
:nmap <DOWN>   :next<CR>
:nmap <UP>     :prev<CR>
```

- Normal maps are also useful for larger commands
- For example:

```
:nmap <silent> ;y : if exists("syntax_on") <BAR>
                  \     syntax off <BAR>
                  \ else <BAR>
                  \     syntax enable <BAR>
                  \ endif<CR>
```

- As you can see `:nmap` is the key to redesigning the `vim` interface to better suit your needs

Command-line maps

- Because they don't require any trailing characters, maps can occasionally be handy for shortcuts on the command-line
- For example, if you frequently write to a backup file:

```
:w ~/backup/latest<CR>
```

- You might prefer:

```
:cmap wb w ~/backup/latest<CR>
```

- Which then allows:

```
:wb
```

Operator-pending maps

- There's a special mapping mode just for operators
- Recall that operators are commands like `c` and `d`
- Commands that expect a motion, or object, or pattern after them
- (As opposed to commands like `R` and `s` that take a count before)
- After you've typed the operator and `vim` is waiting for an operand...
- ...it's in "operator-pending" mode
- You can map key sequences for that special mode with an `:omap`

- For example, if you constantly find yourself wanting `ciw` but typing `cw` instead:

```
:omap w iw
```

- Or if you'd rather `db` mean "delete to end of block", instead of "delete to start of current word":

```
:omap b }
```

Other kinds of maps

- In addition to `:imap` and `:nmap` and `:cmap` and `:omap`, there's also:
- `:xmap`, which defines mappings only for the visual modes
- `:smap`, which defines mappings only for Select mode (a Windows compatible variant on the visual modes)
- `:vmap`, which defines mappings for all the visual modes plus Select mode
- `:map!`, which defines mappings for Insert and Command-line modes
- `:map`, which defines mappings for almost all modes
- `:lmap`, which defines mappings for an incredibly obscure case that you'll almost certainly never encounter or care about
- For more details, see:

```
:help :map-modes
```

- `:vmap` is probably the most useful of these modes
- For example, if the behaviour of `<BS>`/`<DEL>` annoys you in Visual Block mode, you can change it:

```
:vmap <BS> x
```

Managing maps

- You can see the maps you have defined by using the appropriate `map` variant without an argument:

```
:imap
:nmap
```

- You can see a particular mapping by naming it:

```
:imap <TAB>
:nmap v
```

- You can remove a mapping with the appropriate *unmap* variant:

```
:iunmap ww
:iunmap ee
:cunmap wb
```

- You can remove all mappings with the *mapclear* variants:

```
:imapclear
:nmapclear
:vmapclear
```

Unremappable maps

- When you map something, the expansion is re-expanded, if necessary
- That can occasionally be useful:

```
:nmap X dip
:nmap Y XO[Paragraph deleted here]<ESC>:center<CR>
```

- It can even be recursive:

```
:nmap X 0i# <ESC>jX
```

- However, occasionally this re-expansion feature can create problems
- For example, previously we stole Visual mode's command and gave it to Visual Block mode:

```
:nmap v <C-V>
```

- What if we'd wanted to swap the two instead:

```
:nmap    v    <C-V>
:nmap <C-V>    v
```

- Here a `v` would expand to `<CTRL-V>`...
- ...which would re-expand to `v`...
- ...which would re-expand to `<CTRL-V>`...
- ...et cetera, et cetera
- `Vim` detects such infinite recursions and doesn't expand the mapping at all
- To support swapping commands (and other situations where re-expansion is undesirable), `vim` instead provides the *noremap* variants
- For example:

```
:nnoremap    v    <C-V>
:nnoremap <C-V>    v
```

- This causes the RHS of the mapping *not* to be re-expanded
- It's a good (i.e. safe) default

Tip 42: Colour your syntax

- Vim has excellent support for syntax colouring
- Turn it on with:

```
:syntax enable
```

- Turn it off with:

```
:syntax off
```

- Probably best to put them in your `.vimrc`
- Autoselects language, based on the file extension
- Can also manually select:

```
:set syntax=ruby
```

- Supports around 480 languages including: ADA, ANTLR, Apache config, AWK, BASIC, bib, C, changelog, Clipper, COBOL, crontab, csh, CSS, diff, DNS, Doxygen, DTD, Dylan, Eiffel, expect, Forth, Fortran, Foxpro, fstab, Gedcom, gnuplot, groff, Groovy, Haskell, HTML, Icon, Java, JavaScript, lex, Lisp, M4, mail, mailaliases, mailcap, make, man, Maple, Matlab, MySQL, Ocaml, Occam, Pascal, passwd, Perl, PHP, Pod, procmail, Prolog, Python, rcslog, Rexx, robots, Ruby, Samba, SAS, Scheme, sed, SGML, sh, SQL (numerous variants), sshconfig, sshdconfig, SVN, tar, Tcl, tcsh, terminfo, TeX, texinfo, valgrind, Verilog, VHDL, Vim, VRML, xdefaults, XHTML, xinetd, XML, xmodmap, XS, yacc, YAML.

Tip 43: Fix bugs fast

- Optimize the code-compile-debug cycle
- The `:make` command executes your compiler on your buffer then enters "quickfix" mode
- Produces a list of errors that you can step through
- Each step takes you to the location of the next error
- Locations determined by parsing compiler error messages
- By default, `:make` calls *make* and expects back error messages in the (very common) format:

```
<file>:<line>:<error message>
```

- *Vim* comes with parsers for a range of compilers, including gcc, AWK, Jikes, Javac, Ant, Jade, TeX, LaTeX, Pyunit, and Perl
- Can write your own parsers for other compilers' error formats
- Or install a translator for error messages to the default error format

- For example, to debug Perl scripts:

```
:set makeprg=$VIMRUNTIME/tools/efm_perl.pl\ -c\ %\ $*
```

- `efm_perl.pl` converts the usual Perl error messages:

```
<error message> at <file> line <line>
```

- ...to the expected format:

```
<file>:<line>:<error message>
```

Navigating errors

- Once the quickfix error list is built can step through it
- `:cc` *<N>* takes you to the *N*th error
- `:cn` takes you to the next error
- `:cp` takes you to the previous error
- `:cnf` and `:cpf` take you to the first error in the next file or the last error in the previous file

Tip 44: Let `vim` do the indenting for you

- The simplest form of formatting support
- Enable it with:

```
:set autoindent
```

- Now every new line in Insert mode will start at the same column as the previous line
- For more sophisticated indenting, you can also:

```
:set smartindent
```

- Increments the indent when the previous line ends with a `{`
- Decrements the indent when the new line starts with a `}`
- Shunts every shell-like comment (lines starting with #) to the left margin
- That's annoying, so most folks switch it off with:

```
inoremap # X<C-H>#
```

- Smartmatching also increases the indent when previous line starts with any of the words specified by the `cinwords` option
- By default those words are C-like keywords:

```
:set cinwords=if,else,while,do,for,switch
```

- But you can change that:

```
:set cinwords=if,elsif,else,unless,while,until,for,foreach
```

- Of course, if you actually *do* want C-like indenting...
- ...it's best to go all the way:

```
:set cindent
```

- For details of the excruciating range of options available in this mode, see:

```
:help cinoptions
```

Tip 45: Script `vim`

- Vim has its own scripting language
- Fully featured: variables, expressions, variadic argument lists, control structures, built-in functions, user-defined functions, function references, lists, dictionaries, I/O, pattern matching, buffer and window access and control, exceptions, OO, integrated debugger, etc., etc.
- Far too much to cover in detail
- See:

```
:help vim-script-intro
```

- Some examples to illustrate the potential
- You'd put these in your `.vimrc`

```
function! ExpurgateText (text)
    for expletive in g:expletives_list
        a:text = substitute(a:text, expletive, '[DELETED]', 'g')
    endfor
    return a:text
endfunction

function! SaveBackup ()
    execute 'saveas ' . bufname('%') . '.backup_' . g:backup_count
    let g:backup_count += 1
endfunction
```

- Functions can be called either as part of an expression:

```
:let success = setline('.', ExpurgateText(getline('.'))
```

- Or directly via the `:call` command:

```
:call SaveBackup()
```


(damian@conway.org)

- You can make a function aware of the range of lines it's dealing with:

```
function! DeAmp () range
    echo 'DeAmping lines ' . a:firstline . ' to ' . a:lastline
    execute a:firstline . ',' . a:lastline . 's/&/&amp;/g'
endfunction
```

- And then:

```
:.,+10call DeAmp()
```

- The call is applied once and the range passed as implicit arguments

Functions and maps

- Earlier, we saw how to remap the <TAB> character to do completion during insertions:

```
:imap <TAB> <C-N>
```

- That's handy, but so is tabbing
- It would be better if a <TAB> were smart enough to know when it should complete and when it should inserts tabs
- That's easy to achieve by changing the `:imap` to call a function:

```
function! TabOrCompletion()
    let col = col('.') - 1
    if !col || getline('.')[col - 1] !~ '\k'
        return "\<TAB>"
    else
        return "\<C-N>"
    endif
endfunction

:inoremap <silent> <TAB>   <C-R>=TabOrCompletion()<CR>
```

- This uses the <CTRL-R>= Insert-mode command to evaluate an expression and insert the result
- The call to `TabOrCompletion()` examines what's before the cursor and returns either a literal <TAB> or a <CTRL-R> completion request accordingly

Tip 46: Play tag

- Remember the `CTRL-]` feature of the `:help` files?
- That's a specific instance of a much more general facility known as *tagging*
- Can inform `vim` where certain keywords are defined
- In the current file, or in other files
- When you hit `CTRL-]`, `vim` looks up the word under the cursor in a file named `tags` in the current directory
- This file tells `vim` where to jump to
- Theoretically, you could create a "tags" file yourself but that's way too much effort
- Unix systems typically come with a utility named `ctags` which builds tag files for collections of C source files and, in many cases, for collections of source files for other languages
- If your system doesn't provide a `ctags` utility
- ...or your `ctags` doesn't support your chosen development language(s)
- ...you can download and build an open source, multilingual version:

  ```
  http://ctags.sourceforge.net/
  ```

- Can build tag files for Assembler, ASP, Awk, BETA, C, C++, C#, COBOL, Eiffel, Erlang, Fortran, HTML, Java, JavaScript, Lisp, Lua, Make, Pascal, Perl, PHP, PL/SQL, Python, REXX, Ruby, Scheme, Shell scripts (Bourne/Korn/Z), S-Lang, SML, Tcl, Vera, Verilog, Vim, and YACC
- To create a tag file:

  ```
  > ctags *.[ch]
  > ctags *.py
  > ctags *.tcl
  ```

- Once you have a tag file in the current directory any `vim` session will automatically have tags enabled
- In addition, you can tell `vim` where else to search for tags:

  ```
  :set tags+=~/path/to/tag/file/directory/
  ```

- With tags enabled, once you hit `CTRL-]` `vim` looks up the word under the cursor in any available tag files
- It then jumps you to the appropriate file and file
- The typical usage is to place the cursor over a function name and hit `CTRL-]` to be taken to the definition of that function
- Or place the cursor over a constant and hit `CTRL-]`

(damian@conway.org)

- Or place the cursor over a typename and hit `CTRL-]`
- Go back up the tag stack with `CTRL-T`
- If you want to look up an identifier that isn't in the text, use:

```
:tag <identifier>
```

- For example:

```
:tag main
:tag xMalloc
```

- Tags also have completion available:

```
:tag ma<TAB>
```

Tip 47: Have `vim` do some of the coding for you

- `Vim` is event-driven
- Provides hooks that allow you to execute commands automatically whenever particular events occur
- Such commands are specified using the `autocmd` command
- For example, there's a hook for when a new file is created
- Could use it to have `vim` do some set-up for you...

Automatic file skeletons

- Almost every `.h` file has the same skeletal structure
- Almost every `.py` ~~fiel~~ file needs the same basic features
- Almost every Perl documentation specifies the same standard sections
- So it'd be handy if `vim` could set those "boilerplates" up automatically
- Put something like these in your `.vimrc`

```
autocmd BufNewFile *.h    0r ~/templates/skeleton.h
autocmd BufNewFile *.py   0r ~/templates/skeleton.py
autocmd BufNewFile *.pod  0r ~/templates/skeleton.pod
```

- Or you might prefer to have your skeletons generated on-the-fly
- (So they can incorporate edit-time information)
- For example:

```
autocmd BufNewFile *.p[lm] 0r !file_template <afile>
autocmd BufNewFile *.p[lm] 1/^[ \t]*[#].*implementation[
\t]\+here/
```


(damian@conway.org)

Patching files in vim

- Vim has excellent support for applying patches
- ...and examining the effects
- If you have the unpatched file in a buffer, you can simply type:

```
:vert diffpatch <patch_file>
```

- This copies your buffer...
- ...opens a new window beside it with the same contents...
- ...patches the contents of the new window with the patch file
- ...and turns on difference highlighting

Tip 48: Script vim in Perl/Python/Ruby/etc.

- You can script vim in languages other than vimmish
- For example, you could write a simple command that makes a bullet list from a comma'd list:

```
:map ;b :perl
        \ ($line) = $curwin->Cursor;
        \ $curbuf->Append($line, map "\t* $_",
        \           map { /^\s*(.*?)\s*$/ ? $1 : $_ }
        \               split /,\s*/,
        \                   $curbuf->Get($line));
        \ $curbuf->Delete($line)<CR><CR>
```

- There's an API providing access to all windows, buffers, text, options, etc. from within Perl
- For a complete list of the available functions, see:

```
:help :perl-using
```

- If you prefer to script in Python, that's just as easy:

```
:python <<END_PYTHON
    from vim import *
    from string import upper
    current.line = upper(current.line)
END_PYTHON
```

- Vim also has internal interfaces to MzScheme, Tcl, and Ruby

- For more details:

  ```
  :help <language name>
  ```

Tip 49: RTFM

- Often when you're editing code you need to check the manual
- `Vim` makes that easy with the `K` command
- In Normal mode, typing `K` does a `man` on the word under the cursor
- You can change the manual program that's invoked by setting the `keywordprg` option:

  ```
  :set keywordprg=perldoc
  ```

Tip 50: use `vim` as a pager

- Can use `vim` as a replacement for `more` or `less`
- Invoke `vim` via the shell script:

  ```
  $VIMRUNTIME/macros/less.sh
  ```

- Handy because it syntax-highlights
- For example:

  ```
  > $VIMRUNTIME/macros/less.sh ~/Talks/Vim/demo/Main.cc
  ```

Conclusion

- We've covered maybe *one tenth* of the full power of `vim`
- And there many more configuration options, usage variations, extra tricks, et cetera, for the features we have covered
- Other major features include: virtual editing, digraphs, folds, windows, tabbed editing, binary files, text highlighting, active matching, visual quickfix mode, sessions, views, autoformatting of text and comments, spell checking, the "global" command, encryption, mode lines, persistent editing, recording command sequences, printing.
- `:help` is your friend
- `:help <topic><TAB>` doubly so
- You don't have to take immediate advantage of everything we've talked about here
- But there are probably several of these features that would dramatically improve your productivity

- Learn more of the navigational commands
- Try the visual modes when they're more appropriate
- Set the options to help you work the way you like to work
- Create abbreviations and/or maps to short-cut your common tasks
- See if syntax colouring, folds, tags, and quickfixing can facilitate your coding
- There are entire new dimensions of power and convenience hidden behind that familiar `vi` interface
- Explore them!

Appendix A: `vim`'s pattern syntax

- `Vim` uses an extended version of regular expressions
- Not the same as `vi`'s
- Not the same as Perl's (but of comparable power)
- The basic rule is that almost every character matches itself
- With a few exceptions, only backslash-escaped characters are special
- The main features are:

Subpattern...	**Matches...**
`.`	...any character except newline
`*`	...zero or more of the preceding
`^`	...start of line (only at start of pat)
`$`	...end of line (only at end of pat)
`[...]`	...an explicit character class
`\<numerous characters>`	...special behaviour
`\\`	...a literal backslash
`<any other character>`	...itself

- Special behaviours fall into categories as follows...

\(& \) for grouping
\1 \2 for back references

Characters and character classes

- The following backslashed characters are short-hands for one or more "difficult" characters:

Escape	Matches...
\^	literal '^'
\$	literal '$'
_.	any character, including newline
\a	alphabetic character: [A-Za-z]
\A	non-alphabetic character: [^A-Za-z]
\b	<BS>
\d	digit: [0-9]
\D	non-digit: [^0-9]
\e	<ESC>
\f	any character that might appear in a filename
\F	like "\f", but excluding digits
\h	head of word character: [A-Za-z_]
\H	non-head of word character: [^A-Za-z_]
\i	identifier character
\I	like "\i", but excluding digits
\k	keyword character
\K	like "\k", but excluding digits
\l	lowercase character: [a-z]
\L	non-lowercase character: [^a-z]
\n	end-of-line
\o	octal digit: [0-7]
\O	non-octal digit: [^0-7]
\p	printable character
\P	like "\p", but excluding digits
\r	<CR>
\s	whitespace character: <SPACE> and <TAB>
_s	whitespace character: <SPACE>, <TAB>, and newline
\S	non-whitespace character; opposite of \s
\t	<TAB>
\u	uppercase character: [A-Z]

(damian@conway.org)

`\U`	non-uppercase character	`[^A-Z]`
`\w`	word character:	`[0-9A-Za-z_]`
`\W`	non-word character:	`[^0-9A-Za-z_]`
`\x`	hex digit:	`[0-9A-Fa-f]`
`\X`	non-hex digit:	`[^0-9A-Fa-f]`
`\%o<n>`	specified octal character	
`\%d<n>`	specified decimal character	
`\%x<n>`	specified hex character	
`\%u<n>`	specified multibyte character	
`\%U<n>`	specified large multibyte character	

Repetitions

- Zero-or-more: append `*`
- One-or-more: append `\+`
- Zero-or-one: append `\?`
- Exactly-*M*: append `\{M}`
- *M*-to-*N*: append `\{M,N}`
- *M*-or-more: append `\{M,}`
- zero-to-*N*: append `\{,N}`
- For "as few as possible" make first number negative:
- For example, to match a double-quoted string at least one character long:

```
/".\{-1,}"
```

- As a special case of that, `\{-}` minimally matches zero-or-more
- For example, to match everything up to the first occurrence of "__END__":

```
/\_.\{-}__END__
```

Alternatives, synternatives, and sequences

- Alternatives are specified with `\|`
- For example:

```
/perl\|python\|php
```

- "Synternatives" are alternatives where *both* sides have to match
- Specified with a `\&`
- The "and" equivalent of `\|`'s "or"
- For example, to find a line containing the word "Java" and the word "line":

(damian@conway.org)

```
/.*Java\&.*line
```

- *Sequences* are successive characters that may be truncated at any point
- Specified with `\%[...]`
- Very handy when searching for terms that might be abbreviated
- For example:

```
/fun\%[ction]
```

- ...is the same as:

```
/fun\|func\|funct\|functi\|functio\|function
```

Context specifiers

- The ^ and $ markers allow you to constrain where a match can occur
- There are many other such constraint specifiers
- For example, `\_^` and `\_$`, which are the same as ^ and $, except they can appear anywhere in a pattern
- (Handy in alternations and synternations)
- Also `\%^` and `\%$`: start and end of file
- Other positional constraints include:
- ...match only at current cursor position: `\%#`
- ...match only at line *N*: `\%Nl`
- ...match only at column *N*: `\%Nc`
- ...match only at virtual column *N*: `\%Nv` (allowing for tabs)
- Can also put a < or > after the `%` to indicate "before" or "after" the specified row/column
- The \< and \> subpatterns only match at the start/end of a word
- For example:

```
/\<for\>
```

- ...matches "for", but not "fortune" nor "wherefor" nor "enforce"

Match boundaries

- Sometimes you want to use a pattern in a substitution
- You need to match a certain line, but only change part of it
- To make that easy, `vim` provides the `\zs` and `\ze` specifiers
- They alway you to mark where the pattern should be considered to have matched

- Suppose you want to find every call to the function 'update' (provided its first argument starts with a digit) and change that call to a call to 'update_num'
- You could do that with:

```
:%s/\s*\zsupdate\ze \d/update_num/g
```

- The `\zs` and `\ze` tell the substitution that, if it successfully matches the entire pattern...
- ...it should pretend that it only matched from just after the `\zs` to just before the `\ze`
- ...so that the substitution only replaces the "update"

(damian@conway.org)