09/08/12 Edited by: xuzhezhao Last edited: 09/22/12

VIM

学习笔记

Getting Started

1. **vimtutor**

   The vim tutor.

2. **gvim**

   The gvim command causes the editor to create a new window for editing.

3. **vim**

   The editing occurs inside your command window.

4. **Mode**

   The Vim editor is a **modal editor**. That means that the editor behaves differently, depending on which mode you are in.

   To be able to see what mode you are in, type this command:

   :set showmode

5. **Moving around**

h	left	*hjkl*
j	down	
k	up	
l	right	

6. **Finding help**

   Everything you always wanted to know can be found in the Vim help files.

- To get generic help use this command:

:help

- To get help on a given subject, use the following command:

  :help {subject}
- To get help on the "x" command, for example, enter the following:

  :help x
- To find out how to delete text, use this command:

  :help deleting
- To get a complete index of all Vim commands, use the following command:

  :help index
- When you need to get help for a control character command (for example, CTRL-A), you need to spell it with the prefix "CTRL-".

  :help CTRL-A
- When you start the Vim editor, you can use several command-line arguments. These all begin with a dash (-). To find what the -t argument does, for example, use the command:

  :help -t

7. Word movement

w: move the cursor forward one word

b: move backward to the start of the previous word

e: move to the next end of a word

ge: move to the previous end of a word

8. Moving to the start or end of a line

$: moves the cursor to the end of a line

^: moves to the first non-blank character of the line

9. Moving to a characte

f: fx searches forward in the line for the single character x. **[n]fx** is acceptable.

F

t

T

10. Matching a parenthesis

%: When writing a program you often end up with nested () constructs. Then the "%" command is very handy: It moves to the matching paren. If the cursor is on a "(" it will move to the matching ")". If it's on a ")" it will move to the matching "(". This also works for [] and {} pairs. (This can be defined with the 'matchpairs' option.)

11. Moving to a specific line

[n]G

gg

[n]%: For example "50%" moves you to halfway the file. "90%" goes to near the end.

H

M

L

12. Telling where you are

To see where you are in a file, there are three ways.

- Use the **CTRL-G** command (assuming the 'ruler' option is off).

  Sometimes you will see a split column number. For example, "col 2-9". This indicates that the cursor is positioned on the second character, but because character one is a tab, occupying eight spaces worth of columns, the screen column is 9.
- Set the '**number**' option. This will display a line number in front of every line:

  :set number

  To switch this off again:

  :set nonumber
- Set the '**ruler**' option. This will display the cursor position in the lower right corner of the Vim window:

:set ruler

13. Scrolling around

ctrl+u

ctrl+d

ctrl+e

ctrl+f

ctrl+b

ctrl+y

zt

zb

14. Simple searches

To search for a string, use the "**/string**" command.

15. IGNORING CASE

- Normally you have to type exactly what you want to find. If you don't care about upper or lowercase in a word, set the 'ignorecase' option:

  :set ignorecase

- If you now search for "word", it will also match "Word" and "WORD". To match case again:

  :set noignorecase

16. SEARCHING FOR A WORD IN THE TEXT

- Position the cursor on the word and use the "*****" command. Vim will grab the word under the cursor and use it as the search string.
- The "**#**" command does the same in the other direction. You can prepend a count: "**3***" searches for the third occurrence of the word under the cursor.

17. SEARCHING FOR WHOLE WORDS

- If you type "/the" it will also match "there". To only find words that end in "the" use:

  /the\>

  The **"\>"** item is a special marker that only matches at the end of a word. Similarly **"\<"** only matches at the begin of a word. Thus to search for the word "the" only:

  /\<the\>

- This does not match "there" or "soothe". Notice that the "*" and "#" commands use these start-of-word and end-of-word markers to only find whole words (you can use "g*" and "g#" to match partial words).

18. HIGHLIGHTING MATCHES

- **:set hlsearch**

- **:set nohlsearch**

- **:nohlsearch**
  This doesn't reset the option. Instead, it disables the highlighting. As
  soon as you execute a search command, the highlighting will be used again.
  Also for the "n" and "N" commands.

19. TUNING SEARCHES

- **:set incsearch**
  This makes Vim display the **match for the string while you are still typing it.**
  Use this to check if the right match will be found. Then press <Enter> to
  really jump to that location. Or type more to change the search string.
- **:set nowrapscan**
  This stops the search at the end of the file. Or, when you are searching
  backwards, at the start of the file. The 'wrapscan' option is on by default,
  thus searching wraps around the end of the file.

20. INTERMEZZO

- Edit the file, as mentioned at |not-compatible|. Or use this command to
  find out where it is:
  :scriptnames

- Edit the file, for example with:
  :edit ~/.vimrc

- Then add a line with the command to set the option, just like you typed it in
  Vim. Example:
  Go:set hlsearch<Esc>
  "G" moves to the end of the file. **"o"** starts a new line, where you type the
  ":set" command. You end insert mode with <Esc>. Then write the file:

21. Using marks

- When you make a jump to a position with the "**G**" command, Vim remembers the position
  from before this jump. This position is called a **mar**k. To go back where you came from,
  use this command:
  ``

- The **CTRL-O** command jumps to older positions (Hint: O for older). **CTRL-I** then jumps back to newer positions (Hint: I is just next to O on the keyboard).
  Note:
  CTRL-I is the same as <Tab>.

- The "**:jumps**" command gives a list of positions you jumped to. The entry which you used last is marked with a ">".

22. NAMED MARKS

- **m[a~z]**
  `[a~z]

- You can place **26 marks** (a through z) in your text. You can't see them, it's just a position that Vim remembers.

- You can use this command to **get a list of marks**:
  :marks

- You will notice a few **special marks**. These include:
  '' : The cursor position before doing a jump
  " : The cursor position when last editing the file
  [: Start of the last change
  ] : End of the last change

23. Operators and motions

There is a pattern here: **operator-motion**. You first type an **operator command**. For example, "d" is the delete operator. Then you type a **motion command** like "4l" or "w". This way you can operate on any text you can move over.

24. Changing text

Another operator is **"c", change**. It acts just like the "d" operator, except it leaves you in **Insert mode**. For example, "cw" changes a word. Or more specifically, it deletes a word and then puts you in Insert mode.

25. MORE CHANGES

- Like "**dd**" deletes a whole line, "**cc**" changes a whole line. It keeps the existing indent (leading white space) though.
- Just like "d$" deletes until the end of the line, "c$" changes until the end of the line. It's like doing "d$" to delete the text and then "a" to start Insert mode and append new text.

26. SHORTCUTS

Some operator-motion commands are used so often that they have been given a single letter command:

x stands for dl (delete character under the cursor)
X stands for dh (delete character left of the cursor)
D stands for d$ (delete to end of the line)
C stands for c$ (change to end of the line)
s stands for cl (change one character)
S stands for cc (change a whole line)

27. WHERE TO PUT THE COUNT

The commands "**3dw**" and "**d3**w" delete three words. If you want to get really picky about things, the first command, "3dw", **deletes one word three times**; the command "d3w" **deletes three words once**. This is a difference without a distinction. You can actually put in two counts, however. For example, "3d2w" deletes two words, repeated three times, for a total of six words.

28. REPLACING WITH ONE CHARACTER

The **"r"** command is not an operator. It waits for you to type a character, and will replace the character under the cursor with it. You could do the same with "cl" or with the "s" command, but **with "r" you don't have to press <Esc>.**

Using a **count** with "r" causes that many characters to be replaced with **the same character.**

To replace a character with a line break use "r<Enter>". This deletes one character and inserts a line break. Using a count here only applies to the number of characters deleted: "4r<Enter>" replaces four characters with one line break.

29. Repeating a change

- The "." command is one of the most simple yet powerful commands in Vim. It **repeats the last change**. For instance, suppose you are editing an HTML file and want to delete all the

<B> tags. You position the cursor on the first < and delete the <B> with the command "df>". You then go to the < of the next </B> and kill it using the "." command. The "." command executes the last change command (in this case, "df>"). To delete another tag, position the cursor on the < and use the "." command.

- The "." command works for all changes you make, except for the "u" (undo), CTRL-R (redo) and commands that start with a colon (:).

- Another example: You want to change the word "four" to "five". It appears several times in your text. You can do this quickly with this sequence of commands:

```
/four<Enter>    find the first string "four"
cwfive<Esc>     change the word to "five"
n               find the next "four"
.               repeat the change to "five'
n               find the next "four"
.               repeat the change
                etc.
```

30. Visual mode

v

V

ctrl+v

- **GOING TO THE OTHER SIDE**

  If you have selected some text in Visual mode, and discover that you need to change the other end of the selection, use the "o" command (Hint: o for other end). The cursor will go to the other end, and you can move the cursor to change where the selection starts. Pressing "o" again brings you back to the other end.

  When using blockwise selection, you have four corners. "o" only takes you to one of the other corners, diagonally. **Use "O" to move to the other corner in the same line.**

31. Moving text

When you delete something with the "d", "x", or another command, **the text is saved**. You can **paste it back** by using the p command. (The Vim name for this is put).

32. SWAPPING TWO CHARACTERS

Frequently when you are typing, your fingers get ahead of your brain (or the other way around?). The result is a typo such as "teh" for "the". Vim makes it easy to correct such problems. Just put the cursor on the e of "the" and execute the command "**xp**". This works as follows: "x" deletes the character e and places it in a register. "p" puts the text after the cursor, which is after the h.

33. Copying text

The "y" (yank) operator copies text into a register. Then a "p" command can be used to put it.

Since "y" is an operator, you use "yw" to yank a word. A count is possible as usual. To yank two words use "y2w".

34. Using the clipboard

- If you are not using the GUI, or if you don't like using a menu, you have to use another way. You use the normal "y" (yank) and "p" (put) commands, but **prepend "*** (double-quote star) before it. To copy a line to the clipboard:

  "*yy

- To put text from the clipboard back into the text:

  "*p

35. Text objects

- If the cursor is in the middle of a word and you want to delete that word, you need to move back to its start before you can do "dw". There is a simpler way to do this: "**daw**".

- Hint: **"aw" stands for "A Word"**. Thus "daw" is "Delete A Word". To be precise, the white space after the word is also deleted (the white space before the word at the end of the line).

- To change a whole sentence use "**cis**". Take this text:

- "cis" consists of the "c" (change) operator and the "is" text object. This stands for "Inner Sentence". There is also the "as" (a sentence) object. The difference is that "as" includes the white space after the sentence and "is" doesn't. If you would delete a sentence, you want to delete the white space at the same time, thus use "das". If you want to type new text the white space can remain, thus you use "cis".

36. Replace mode

The "R" command causes Vim to enter replace mode. In this mode, each character you type replaces the one under the cursor. This continues until you type <Esc>.

When you use <BS> (backspace) to make correction, you will notice that the old text is put back. Thus it works like an undo command for the last typed character.

37. The vimrc file

You probably got tired of typing commands that you use very often. To start Vim with all your favorite option settings and mappings, you write them in what is called the vimrc file. Vim executes the commands in this file when it starts up.

- If you already have a vimrc file (e.g., when your sysadmin has one setup for you), you can edit it this way:

  :edit $MYVIMRC

- The vimrc file can contain all the commands that you type after a colon. The most simple ones are for setting options.

38. The example vimrc file explained

- In the first chapter was explained how the example vimrc (included in the Vim distribution) file can be used to make Vim startup in not-compatible mode (see |not-compatible|). The file can be found here:

  $VIMRUNTIME/vimrc_example.vim

- In this section we will explain the various commands used in this file. This will give you hints about how to set up your own preferences. Not everything will be explained though. Use the "**:help**" command to find out more.

- **set nocompatible**
  As mentioned in the first chapter, these manuals explain Vim working in an improved way, thus not completely Vi compatible. Setting the 'compatible' option off, thus 'nocompatible' takes care of this.

- **set backspace=indent,eol,start**
  This specifies where in Insert mode the <BS> is allowed to delete the character in front of

the cursor. The three items, separated by commas, tell Vim to delete the white space at the start of the line, a line break and the character before where Insert mode started.

- **set autoindent**
  This makes Vim use the **indent** of the previous line for a newly created line. Thus there is the same amount of white space before the new line. For example when pressing <Enter> in Insert mode, and when using the "o" command to open a new line.

- **if has("vms")**
  set nobackup
  else
  set backup
  endif

  This tells Vim to keep a backup copy of a file when overwriting it. But not on the VMS system, since it keeps old versions of files already. The backup file will have the same name as the original file with "~" added.

- **set history=50**

  Keep 50 commands and 50 search patterns in the history. Use another number if you want to remember fewer or more lines.

- **set ruler**

  Always display the current cursor position in the lower right corner of the Vim window.

- **set showcmd**

  Display an incomplete command in the lower right corner of the Vim window, left of the ruler. For example, when you type "2f", Vim is waiting for you to type the character to find and "2f" is displayed. When you press "w" next, the "2fw" command is executed and the displayed "2f" is removed.

- **set incsearch**
  Display the match for a search pattern when halfway typing it.

- **map Q gq**

  This defines a key mapping. More about that in the next section. This defines the "Q" command to do formatting with the "gq" operator. This is how it worked before Vim 5.0. Otherwise the "Q" command starts Ex mode, but you will not need it.

- **vnoremap _g y:exe "grep /" . escape(@", '\\/') . "/ *.c *.h"<CR>**

  This mapping yanks the visually selected text and searches for it in C files. This is a complicated mapping. You can see that mappings can be used to do quite complicated things. Still, it is just a sequence of commands that are executed like you typed them.

-
  ```
  if &t_Co > 2 || has("gui_running")
      syntax on
      set hlsearch
  endif
  ```

  This switches on syntax highlighting, but only if colors are available. And the 'hlsearch' option tells Vim to highlight matches with the last used search pattern. The "if" command is very useful to set options only when some condition is met.

vimrc-filetype

- **filetype plugin indent on**

  This switches on three very clever mechanisms:

  1. Filetype detection.

  Whenever you start editing a file, Vim will try to figure out what kind of file this is. When you edit "main.c", Vim will see the ".c" extension and recognize this as a "c" filetype. When you edit a file that starts with "#!/bin/sh", Vim will recognize it as a "sh" filetype.
  The filetype detection is used for syntax highlighting and the other two items below.

  2. Using filetype plugin files

  Many different filetypes are edited with different options. For example, when you edit a "c" file, it's very useful to set the 'cindent' option to automatically indent the lines. These commonly useful option settings are included with Vim in filetype plugins. You can also add your own.

  3. Using indent files

When editing programs, the indent of a line can often be computed automatically. Vim comes with these indent rules for a number of filetypes. See |:filetype-indent-on| and 'indentexpr'.

- **autocmd FileType text setlocal textwidth=78**

  This makes Vim break text to avoid lines getting longer than 78 characters. But only for files that have been detected to be plain text. There are actually two parts here. "autocmd FileType text" is an autocommand. This defines that when the file type is set to "text" the following command is automatically executed. "setlocal textwidth=78" sets the 'textwidth' option to 78, but only locally in one file.

restore-cursor

- **autocmd BufReadPost ***

```
    \ if line("'\"") > 1 && line("'\"") <= line("$") |
    \   exe "normal! g`\"" |
    \ endif
```

  Another autocommand. This time it is used after reading any file. The complicated stuff after it checks if the '"' mark is defined, and jumps to it if so. The backslash at the start of a line is used to continue the command from the previous line. That avoids a line getting very long. See |line-continuation|. This only works in a Vim script file, not when typing commands at the command-line.

39. Simple mappings

A **mapping** enables you to **bind a set of Vim commands to a single key**. Suppose, for example, that you need to surround certain words with curly braces. In other words, you need to change a word such as "amount" into "{amount}". With the **:map** command, you can tell Vim that the F5 key does this job. The command is as follows:

```
:map <F5> i{<Esc>ea}<Esc>
```

Note:

When entering this command, you must enter <F5> by typing four characters. Similarly, <Esc> is not entered by pressing the <Esc> key, but by typing five characters.

- In this example, the trigger is a single key; it can be any string. But when you use an existing Vim command, that command will no longer be available. You better avoid that.

  One key that can be used with mappings is the backslash. Since you probably want to define more than one mapping, add another character. You could map "\p" to add parentheses around a word, and "\c" to add curly braces, for example:

```
    :map \p i(<Esc>ea)<Esc>
    :map \c i{<Esc>ea}<Esc>
```

- The "**:map**" command (with no arguments) lists your current mappings. At least the ones for Normal mode. More about mappings in section |40.1|.

40. Adding a plugin

Vim's functionality can be extended by **adding plugins**. A plugin is nothing more than a Vim script file that is loaded automatically when Vim starts. You can add a plugin very easily by dropping it in your plugin directory. {not available when Vim was compiled without the |+eval| feature}

There are two types of plugins:
global plugin: Used for all kinds of files
filetype plugin: Only used for a specific type of file

41. GLOBAL PLUGINS

When you start Vim, it will automatically load a number of **global plugins**. You don't have to do anything for this. They add functionality that most people will want to use, but which was implemented as a **Vim script** instead of being compiled into Vim. You can find them listed in the help index |standard-plugin-list|. Also see |load-plugins|.

- **Add-global-plugin**

You can add a global plugin to add functionality that will always be present
when you use Vim. There are only two steps for adding a global plugin:
1. Get a copy of the plugin.
2. Drop it in the right directory.

42. GETTING A GLOBAL PLUGIN

Where can you find plugins?
- Some come with Vim. You can find them in the directory **$VIMRUNTIME/macros** and its sub-directories.
- Download from the net. There is a large collection on http://www.vim.org.
- They are sometimes posted in a Vim |maillist|.
- You could write one yourself, see |write-plugin|.

Some plugins come as a vimball archive, see |vimball|.
Some plugins can be updated automatically, see |getscript|.

43. USING A GLOBAL PLUGIN

First read the text in the plugin itself to check for any special conditions.
Then copy the file to your plugin directory:

system	**plugin directory**
Unix	**~/.vim/plugin/**
PC and OS/2	$HOME/vimfiles/plugin or $VIM/vimfiles/plugin
Amiga	s:vimfiles/plugin
Macintosh	$VIM:vimfiles:plugin
Mac OS X	~/.vim/plugin/
RISC-OS	Choices:vimfiles.plugin

Example for Unix (assuming you didn't have a plugin directory yet):

```
mkdir ~/.vim
mkdir ~/.vim/plugin
cp /usr/local/share/vim/vim60/macros/justify.vim ~/.vim/plugin
```

Notes: Instead of putting plugins directly into the plugin/ directory, you may better organize them by putting them into **subdirectories** under plugin/. As an example, consider using "~/.vim/plugin/perl/*.vim" for all your Perl plugins.

44. FILETYPE PLUGINS

The Vim distribution comes with a set of plugins for different filetypes that you can start using with this command:

```
:filetype plugin on
```

- **add-filetype-plugin**

  If you are missing a plugin for a filetype you are using, or you found a better one, you can add it. There are two steps for adding a filetype plugin:
  1. Get a copy of the plugin.
  2. Drop it in the right directory.

45. GETTING A FILETYPE PLUGIN

You can find them in the same places as the global plugins. Watch out if the type of file is mentioned, then you know if the plugin is a global or a filetype one. The scripts in $VIMRUNTIME/macros are global ones, the filetype plugins are in **$VIMRUNTIME/ftplugin**.

46. USING A FILETYPE PLUGIN

You can add a filetype plugin by dropping it in the right directory. The name of this directory is in the same directory mentioned above for global plugins, but the last part is "ftplugin". Suppose you have found a plugin for the "stuff" filetype, and you are on Unix. Then you can move this file to the ftplugin directory:

```
mv thefile ~/.vim/ftplugin/stuff.vim
```

If that file already exists you already have a plugin for "stuff". You might want to check if the existing plugin doesn't conflict with the one you are adding. If it's OK, you can give the new one another name:

```
mv thefile ~/.vim/ftplugin/stuff_too.vim
```

The underscore is used to separate the name of the filetype from the rest, which can be anything. If you use **"otherstuff.vim" it wouldn't work**, it would be loaded for the **"otherstuff" filetype.**

- **The generic names for the filetype plugins are:**

```
ftplugin/<filetype>.vim
ftplugin/<filetype>_<name>.vim
ftplugin/<filetype>/<name>.vim
```

The **<filetype>** part is the name of the filetype the plugin is to be used for. Only files of this filetype will use the settings from the plugin. The <name> part of the plugin file doesn't matter, you can use it to have several plugins for the same filetype. Note that it must **end in ".vim".**

47. Adding a help file

Now create a "doc" directory in one of the directories in 'runtimepath'.

```
:!mkdir ~/.vim/doc
```

Copy the help file to the "doc" directory.

```
:!cp $VIMRUNTIME/macros/matchit.txt ~/.vim/doc
```

Now comes the trick, which allows you to jump to the subjects in the new help file: Generate the local tags file with the |:helptags| command.

```
:helptags ~/.vim/doc
```

Now you can use the

```
:help g%
```

command to find help for "g%" in the help file you just added. You can see an entry for the local help file when you do:

```
:help local-additions
```

The title lines from the local help files are automagically added to this section. There you can see which local help files have been added and jump to them through the tag.

48. The option window

- If you are looking for an option that does what you want, you can search in the help files here: |options|. Another way is by using this command:

  ```
  :options
  ```

- This opens a new window, with a list of options with a one-line explanation. The options are grouped by subject. Move the cursor to a subject and press <Enter> to jump there. Press <Enter> again to jump back. Or use **CTRL-O**.

- You can change the value of an option. For example, move to the "displaying text" subject. Then move the cursor down to this line:

  ```
  set wrap      nowrap
  ```

  When you hit <Enter>, the line will change to:

  ```
  set nowrap    wrap
  ```

  The option has now been switched off.

- Just above this line is a short description of the 'wrap' option. Move the cursor one line up to place it in this line. Now hit <Enter> and you jump to the full help on the 'wrap' option.

For options that take a number or string argument you can edit the value.
Then press <Enter> to apply the new value. For example, move the cursor a few lines up to this line:

set so=0

Position the cursor on the zero with "$". Change it into a five with "r5". Then press <Enter> to apply the new value. When you now move the cursor around you will notice that the text starts scrolling before you reach the border. This is what the 'scrolloff' option does, it specifies an offset from the window border where scrolling starts.

49. Often used options

There are an awful lot of options. Most of them you will hardly ever use. Some of the more useful ones will be mentioned here. Don't forget you can find more help on these options with the ":help" command, with single quotes before and after the option name. For example:

:help 'wrap'

- In case you have messed up an option value, you can set it back to the default by putting an ampersand (&) after the option name. Example:

  :set iskeyword&

50. NOT WRAPPING LINES

- Vim normally wraps long lines, so that you can see all of the text. Sometimes it's better to let the text continue right of the window. Then you need to scroll the text left-right to see all of a long line. Switch wrapping off with this command:

  :set nowrap

- Vim will automatically scroll the text when you move to text that is not displayed. To see a context of ten characters, do this:

  :set sidescroll=10

  This doesn't change the text in the file, only the way it is displayed.

- Most commands for moving around will stop moving at the start and end of a line. You can change that with the 'whichwrap' option. This sets it to the default value:

  :set whichwrap=b,s

  This allows the <BS> key, when used in the first position of a line, to move the cursor to the end of the previous line. And the <Space> key moves from the end of a line to the start of the next one.

- To allow the cursor keys <Left> and <Right> to also wrap, use this command:
  ```
  :set whichwrap=b,s,<,>
  ```

- This is still only for Normal mode. To let <Left> and <Right> do this in Insert mode as well:
  ```
  :set whichwrap=b,s,<,>,[,]
  ```

VIEWING TABS

- When there are tabs in a file, you cannot see where they are. To make them visible:
  ```
  :set list
  ```

  Now every tab is displayed as ^I. And a $ is displayed at the end of each line, so that you can spot trailing spaces that would otherwise go unnoticed.

  A disadvantage is that this looks ugly when there are many Tabs in a file. If you have a color terminal, or are using the GUI, Vim can show the spaces and tabs as highlighted characters. Use the 'listchars' option:
  ```
  :set listchars=tab:>-,trail:-
  ```

  Now every tab will be displayed as ">---" (with more or less "-") and trailing white space as "-". Looks a lot better, doesn't it?

KEYWORDS

- The '**iskeyword**' option specifies which characters can appear in a word:
  ```
  :set iskeyword
    iskeyword=@,48-57,_,192-255
  ```

  The "@" stands for all alphabetic letters. "48-57" stands for ASCII characters 48 to 57, which are the numbers 0 to 9. "192-255" are the printable latin characters.

  Sometimes you will want to **include a dash in keywords**, so that commands like "w" consider "upper-case" to be one word. You can do it like this:
  ```
  :set iskeyword+=-
  :set iskeyword
    iskeyword=@,48-57,_,192-255,-
  ```

  If you look at the new value, you will see that Vim has added a comma for you.

  To **remove a character** use "-=". For example, to remove the underscore:
  ```
  :set iskeyword-=_
  ```

```
:set iskeyword
  iskeyword=@,48-57,192-255,-
```

This time a comma is automatically deleted.

ROOM FOR MESSAGES

- When Vim starts there is one line at the bottom that is used for messages. When a message is long, it is either truncated, thus you can only see part of it, or the text scrolls and you have to press <Enter> to continue. You can set the 'cmdheight' option to the **number of lines used for messages**. Example:

  ```
  :set cmdheight=3
  ```

  This does mean there is less room to edit text, thus it's a compromise.

Using syntax highlighting

51. Switching it on

- It all starts with one simple command:

  ```
  :syntax enable
  ```

- That should work in most situations to get color in your files. Vim will automagically detect the type of file and load the right syntax highlighting. Suddenly comments are blue, keywords brown and strings red. This makes it easy to overview the file. After a while you will find that black&white text slows you down!

- If you want syntax highlighting only when the terminal supports colors, youcan put this in your |vimrc| file:

  ```
  if &t_Co > 1
    syntax enable
  endif
  ```

- If you want syntax highlighting only in the GUI version, put the ":syntax enable" command in your |**gvimrc**| file.

52. Different colors

If you don't like the default colors, you can select another color scheme. In the GUI use the Edit/Color Scheme menu. You can also type the command:

```
:colorscheme evening
```

"evening" is the name of the color scheme. There are several others you might want to try out. Look in the directory **$VIMRUNTIME/colors**.

53. With colors or without colors

Displaying text in color takes a lot of effort. If you find the displaying too slow, you might want to disable syntax highlighting for a moment:

```
:syntax clear
```

When editing another file (or the same one) the colors will come back.

If you want to stop highlighting completely use:

```
:syntax off
```

This will completely disable syntax highlighting and remove it immediately for all buffers.

If you want syntax highlighting only for specific files, use this:

```
:syntax manual
```

This will enable the syntax highlighting, but not switch it on automatically when starting to edit a buffer. To switch highlighting on for the current buffer, set the 'syntax' option:

```
:set syntax=ON
```

54. Printing with colors

- In the **MS-Windows** version you can print the current file with this command:

  ```
  :hardcopy
  ```

  There are several options that change the way Vim prints:

  'printdevice'
  'printheader'
  'printfont'
  'printoptions'

- This also works on **Unix**, if you have a PostScript printer. Otherwise, you will have to do a bit more work. You need to convert the text to HTML first, and then print it from a web browser.

- Convert the current file to HTML with this command:

  :TOhtml

  In case that doesn't work:

  :source $VIMRUNTIME/syntax/2html.vim

Editing more than one file

55. Edit another file

:edit foo.txt

Note:

Vim puts an error ID at the start of each error message. If you do not understand the message or what caused it, look in the help system for this ID. In this case:

:help E37

Or you can force Vim to discard your changes and edit the new file, using the force (!) character:

:edit! foo.txt

If you want to edit another file, but not write the changes in the current file yet, you can make it hidden:

:hide edit foo.txt

56. A list of files

vim one.c two.c three.c

:next

:next!

:wnext

If you want to see the list of files, use this command:

:args

57. MOVING TO OTHER ARGUMENTS

:previous

:wprevious

:last

:first

:2next

58. AUTOMATIC WRITING

When moving around the files and making changes, you have to remember to use ":write". Otherwise you will get an error message. If you are sure you always want to write modified files, you can tell Vim to automatically write them:

:set autowrite

When you are editing a file which you may not want to write, switch it off again:

:set noautowrite

59. EDITING ANOTHER LIST OF FILES

You can redefine the list of files without the need to exit Vim and start it again. Use this command to edit three other files:

:args five.c six.c seven.h

Or use a wildcard, like it's used in the shell:

:args *.txt

Vim will take you to the first file in the list. Again, if the current file has changes, you can either write the file first, or use "**:args!**" (with ! added) to abandon the changes.

60. DID YOU EDIT THE LAST FILE?

When you use a list of files, Vim assumes you want to edit them all. To protect you from exiting too early, you will get this error when you didn't edit the last file in the list yet:

E173: 46 more files to edit

If you really want to exit, just do it again. Then it will work (but not when you did other commands in between).

61. Jumping from file to file

To quickly jump between two files, press **CTRL-^** (on English-US keyboards the ^ is above the 6 key). Example:

:args one.c two.c three.c

You are now in one.c.

:next

Now you are in two.c. Now use CTRL-^ to go back to one.c. Another CTRL-^ and you are back in two.c. Another CTRL-^ and you are in one.c again. If you now do:

:next

You are in three.c. Notice that the CTRL-^ command **does not change** the idea of **where you are** in the list of files. **Only commands like ":next" and ":previous" do that.**

The file you were previously editing is called the "**alternate" fil**e. When you just started Vim **CTRL-^** will not work, since there isn't a previous file.

62. PREDEFINED MARKS

After jumping to another file, you can use two predefined marks which are very useful:

`"

This takes you to the position where the cursor was when you left the file.

Another mark that is remembered is the position where you made the last change:

`.

Suppose you are editing the file "one.txt". Somewhere halfway the file you use "x" to delete a character. Then you go to the last line with "G" and write the file with ":w". You edit several other files, and then use ":edit one.txt" to come back to "one.txt". If you now use `" Vim jumps to the last line of the file. Using `. takes you to the position where you deleted the character. Even when you move around in the file `" and `. will take you to the remembered position. At least until you make another change or leave the file.

63. FILE MARKS

In chapter 4 was explained how you can place a mark in a file with "mx" and jump to that position with "`x". That works within one file. If you edit another file and place marks there, these are specific for that file. Thus each file has its own set of marks, they are **local to the file.**

So far we were using marks with a lowercase letter. There are also marks with an

uppercase letter. These are **global**, they can be used from **any file**. For example suppose that we are editing the file "foo.txt". Go to halfway the file ("50%") and place the F mark there (F for foo):

```
50%mF
```

Now edit the file "bar.txt" and place the B mark (B for bar) at its last line:

```
GmB
```

Now you can use the "'F" command to jump back to halfway foo.txt. Or edit yet another file, type "'B" and you are at the end of bar.txt again.

The file marks are remembered until they are placed somewhere else. Thus you can place the mark, do hours of editing and still be able to jump back to that mark.

It's often useful to think of a simple connection between the mark letter and where it is placed. For example, use the H mark in a header file, M in a Makefile and C in a C code file.

To see where a specific mark is, give an argument to the ":marks" command:

```
:marks M
```

You can also give several arguments:

```
:marks MCP
```

Don't forget that you can use **CTRL-O** and **CTRL-I** to jump to older and newer positions without placing marks there.

64. Backup files

Usually Vim does not produce a backup file. If you want to have one, all you need to do is execute the following command:

```
:set backup
```

If you do not like the fact that the backup files end with ~, you can change the extension:

```
:set backupext=.bak
```

Another option that matters here is '**backupdir**'. It specifies where the backup file is written. The default, to write the backup in the same directory as the original file, will mostly be the right thing.

Note:

When the **'backup' option isn't set** but the '**writebackup**' is, Vim will still create a backup file. However, it is deleted as soon as writing the file was completed successfully. This functions as a safety against losing your original file when writing fails in some way (disk full is the most common cause; being hit by lightning might be another, although less common).

65. KEEPING THE ORIGINAL FILE

If you are editing source files, you might want to keep the file before you make any changes. But the backup file will be overwritten each time you write the file. Thus it only contains the previous version, not the first one. To make Vim keep the original file, set the 'patchmode' option. This specifies the extension used for the first backup of a changed file. Usually you would do this:

```
:set patchmode=.orig
```

When you now edit the file data.txt for the first time, make changes and write the file, Vim will keep a copy of the unchanged file under the name "**data.txt.orig**".

If you make further changes to the file, Vim will notice that "data.txt.orig" already exists and leave it alone. Further backup files will then be called "data.txt~" (or whatever you specified with 'backupext').

If you leave 'patchmode' empty (that is the default), the original file will not be kept.

66. Copy text between files

67. USING REGISTERS

When you want to copy several pieces of text from one file to another, having to switch between the files and writing the target file takes a lot of time. To avoid this, copy each piece of text to its own **register**.

A register is a place where Vim stores text. Here we will use the registers named **a to z** (later you will find out there are others). Let's copy a sentence to the f register (f for First):

```
"fyas
```

The "yas" command yanks a sentence like before. It's the **"f** that tells Vim the text should be place in the f register. This must come just before the yank command.

Now yank three whole lines to the l register (l for line):

```
"l3Y
```

The count could be before the "l just as well. To yank a block of text to the **b (for block) register:**

CTRL-Vjjww"by

Notice that the register specification **"b** is just before the "y" command. This is required. If you would have put it before the "w" command, it would not have worked.

Now you have three pieces of text in the f, l and b registers. Edit another file, move around and place the text where you want it:

"fp

Again, the register specification "f comes before the "p" command.

You can put the registers in any order. And the text stays in the register until you yank something else into it. Thus you can put it as many times as you like.

When you **delete text**, you can also specify a register. Use this to move several pieces of text around. For example, to delete-a-word and write it in the w register:

"wdaw

Again, the register specification comes before the delete command "d".

68. APPENDING TO A FILE

When collecting lines of text into one file, you can use this command:

:write >> logfile

To append only a few lines, select them in Visual mode before typing ":write". In chapter 10 you will learn other ways to select a range of lines.

69. Viewing a file

Sometimes you only want to see what a file contains, without the intention to ever write it back. There is the risk that you type ":w" without thinking and overwrite the original file anyway. To avoid this, edit the file read-only.

To start Vim in readonly mode, use this command:

vim -R file

On Unix this command should do the same thing:

view file

If you make changes to a file and forgot that it was read-only, you can still write it. Add the ! to the write command to force writing.

If you really want to **forbid making changes** in a file, do this:

```
vim -M file
```

Now every attempt to change the text will fail. The help files are like this, for example. If you try to make a change you get this error message:

```
E21: Cannot make changes, 'modifiable' is off
```

You could use the -M argument to setup Vim to work in a viewer mode. This is only voluntary though, since these commands will remove the protection:

```
:set modifiable
:set write
```

70. Changing the file name

A clever way to start editing a new file is by using an existing file that contains most of what you need. For example, you start writing a new program to move a file. You know that you already have a program that copies a file, thus you start with:

```
:edit copy.c
```

You can delete the stuff you don't need. Now you need to save the file under a new name. The ":saveas" command can be used for this:

```
:saveas move.c
```

Vim will write the file under the given name, and edit that file. Thus the next time you do ":write", it will write "move.c". "copy.c" remains unmodified.

When you want to change the name of the file you are editing, but don't want to write the file, you can use this command:

```
:file move.c
```

Vim will mark the file as "not edited". This means that Vim knows this is not the file you started editing. When you try to write the file, you might get this message:

```
E13: File exists (use ! to override)
```

This protects you from **accidentally overwriting another file.**

Splitting windows

71. Split a window

The easiest way to open a new window is to use the following command:

```
:split
```

This command splits the screen into two windows and leaves the cursor in the top one:

The two windows allow you to view two parts of the same file. For example, you could make the top window show the variable declarations of a program, and the bottom one the code that uses these variables.

The **CTRL-W w** command can be used to jump between the windows. If you are in the top window, **CTRL-W w** jumps to the window below it. If you are in the bottom window it will jump to the first window. (**CTRL-W CTRL-W** does the same thing, in case you let go of the CTRL key a bit later.)

72. CLOSE THE WINDOW

To close a window, use the command:

```
:close
```

Actually, any command that quits editing a file works, like ":quit" and "ZZ". But ":close" prevents you from accidentally exiting Vim when you close the last window.

73. CLOSING ALL OTHER WINDOWS

If you have opened a whole bunch of windows, but now want to concentrate on one of them, this command will be useful:

```
:only
```

This closes all windows, except for the current one. If any of the other windows has changes, you will get an error message and that window won't be closed.

74. Split a window on another file

The following command opens a second window and starts editing the given file:

```
:split two.c
```

To open a window on a new, empty file, use this:

    :new

75. Window size

The "**:split**" command can take a number argument. If specified, this will be the height of the new window. For example, the following opens a new window three lines high and starts editing the file alpha.c:

    :3split alpha.c

To increase the size of a window:

    CTRL-W +

To decrease it:

    CTRL-W -

Both of these commands take a count and increase or decrease the window size by that many lines. Thus "**4 CTRL-W +**" make the window four lines higher.

To set the window height to a specified number of lines:

    {height}CTRL-W _

To make a window as high as it can be, use the CTRL-W _ command without a count.

76. OPTIONS

The **'winheight'** option can be set to a minimal desired height of a window and **'winminheight'** to a hard minimum height.

Likewise, there is **'winwidth'** for the minimal desired width and 'winminwidth' for the hard minimum width.

The **'equalalways'** option, when set, makes Vim equalize the windows sizes when a window is closed or opened.

77. Vertical splits

The ":split" command creates the new window above the current one. To make the window appear at the left side, use:

:vsplit

or:

:vsplit two.c

There is also the "**:vnew**" command, to open a vertically split window on a new, empty file. Another way to do this:

:vertical new

The "**:vertical**" command can be inserted before another command that splits a window. This will cause that command to split the window vertically instead of horizontally. (If the command doesn't split a window, it works unmodified.)

78. MOVING BETWEEN WINDOWS

CTRL-W h	move to the window on the left
CTRL-W j	move to the window below
CTRL-W k	move to the window above
CTRL-W l	move to the window on the right
CTRL-W t	move to the TOP window
CTRL-W b	move to the BOTTOM window

79. Moving windows

CTRL-W K	**m**ove window to the far top
CTRL-W H	move window to the far left
CTRL-W J	move window to the bottom
CTRL-W L	move window to the far right

Note: This uses the uppercase letter K H J L.

80. Commands for all windows

:qall

:wall

:wqall

:qall!

81. OPENING A WINDOW FOR ALL ARGUMENTS

To make Vim open a window for each file, start it with the "-o" argument:

vim -o one.txt two.txt three.txt

The **"-O"** argument is used to get vertically split windows.
When Vim is already running, the **":all"** command opens a window for each file in the argument list. **":vertical all"** does it with vertical splits.

82. Viewing differences with vimdiff

```
vimdiff main.c~ main.c
```

83. THE FOLD COLUMN

84. DIFFING IN VIM

Another way to start in diff mode can be done from inside Vim. Edit the "main.c" file, then make a split and show the differences:

```
:edit main.c
:vertical diffsplit main.c~
```

If you have a patch or diff file, you can use the third way to start diff mode. First edit the file to which the patch applies. Then tell Vim the name of the patch file:

```
:edit main.c
:vertical diffpatch main.c.diff
```

WARNING: The patch file must contain only one patch, for the file you are editing. Otherwise you will get a lot of error messages, and some files might be patched unexpectedly.
The patching will only be done to the copy of the file in Vim. The file on your harddisk will remain unmodified (until you decide to write the file).

85. SCROLL BINDING

When the files have more changes, you can scroll in the usual way. Vim will try to keep both the windows start at the same position, so you can easily see the differences side by side.
When you don't want this for a moment, use this command:

```
:set noscrollbind
```

86. JUMPING TO CHANGES

When you have disabled folding in some way, it may be difficult to find the changes. Use this command to jump forward to the next change:

```
]c
```

To go the other way use:

```
[c
```

Prepended a count to jump further away.

87. REMOVING CHANGES

You can move text from one window to the other. This either removes differences or adds new ones. Vim doesn't keep the highlighting updated in all situations. To update it use this command:

```
:diffupdate
```

To remove a difference, you can move the text in a highlighted block from one window to another. Take the "main.c" and "main.c~" example above. Move the cursor to the left window, on the line that was deleted in the other window. Now type this command:

```
dp
```

The change will be removed by putting the text of the current window in the other window. "dp" stands for "diff put".

You can also do it the other way around. Move the cursor to the right window, to the line where "changed" was inserted. Now type this command:

```
do
```

The change will now be removed by getting the text from the other window. Since there are no changes left now, Vim puts all text in a closed fold. "do" stands for "diff obtain". "dg" would have been better, but that already has a different meaning ("dgg" deletes from the cursor until the first line).

For details about diff mode, see |vimdiff|.

88. Various

The 'laststatus' option can be used to specify when the last window has a statusline:

0	never

1	only when there are split windows (the default)
2	always

Many commands that edit another file have a variant that splits the window. For Command-line commands this is done by prepending an "s". For example: ":tag" jumps to a tag, ":stag" splits the window and jumps to a tag.
For Normal mode commands a CTRL-W is prepended. CTRL-^ jumps to the alternate file, CTRL-W CTRL-^ splits the window and edits the alternate file.

The 'splitbelow' option can be set to make a new window appear below the current window. The 'splitright' option can be set to make a vertically split window appear right of the current window.

When splitting a window you can prepend a modifier command to tell where the window is to appear:

:leftabove {cmd}	left or above the current window
:aboveleft {cmd}	idem
:rightbelow {cmd}	right or below the current window
:belowright {cmd}	idem
:topleft {cmd}	at the top or left of the Vim window
:botright {cmd}	at the bottom or right of the Vim window

89. Tab pages

You will have noticed that windows never overlap. That means you quickly run out of screen space. The solution for this is called Tab pages.

Assume you are editing "thisfile". To create a new tab page use this command:

```
:tabedit thatfile
```

If you don't have a mouse or don't want to use it, you can use the **"gt"** command. Mnemonic: **Goto Tab**.

Now let's create another tab page with the command:

```
:tab split
```

You can put ":tab" before any Ex command that opens a window. The window will be opened in a new tab page. Another example:

```
:tab help gt
```

Using the GUI

Vim works in an ordinary terminal. GVim can do the same things and a few more. The GUI offers menus, a toolbar, scrollbars and other items. This chapter is about these extra things that the GUI offers.

90. Parts of the GUI

91. THE TOOLBAR

If you never want a toolbar, use this command in your vimrc file:

```
:set guioptions-=T
```

Making big changes

In chapter 4 several ways to make small changes were explained. This chapter goes into making changes that are repeated or can affect a large amount of text. The Visual mode allows doing various things with blocks of text. Use an external program to do really complicated things.

92. Record and playback commands

The **"."** command repeats the preceding change. But what if you want to do something more complex than a single change? That's where command recording comes in. There are three steps:

1. The **"q{register}"** command starts recording keystrokes into the register named **{register}**. The register name must be between **a and z**.
2. Type your commands.
3. To finish recording, press **q** (without any extra character). You start by moving to the first character of the first line. Next you execute the following commands:

Take a look at how to use these commands in practice. You have a list of filenames that look like this:

stdio.h
fcntl.h
unistd.h
stdlib.h

And what you want is the following:

#include "stdio.h"
#include "fcntl.h"
#include "unistd.h"
#include "stdlib.h"

qa	Start recording a macro in register a.
^	Move to the beginning of the line.
i#include "<Esc>	Insert the string #include " at the beginning of the line.
$	Move to the end of the line.
a"<Esc>	Append the character double quotation mark (") to the end of the line.
j	Go to the next line.
q	Stop recording the macro.

Now that you have done the work once, you can repeat the change by typing the command "@a" three times.
The "@a" command can be preceded by a count, which will cause the macro to be executed that number of times. In this case you would type:
3@a

93. MOVE AND EXECUTE

You might have the lines you want to change in various places. Just move the cursor to each location and use the "@a" command. If you have done that once, you can do it again with "**@@**". That's a bit easier to type. **If you now execute register b with "@b", the next "@@" will use register b.**
If you compare the playback method with using ".", there are several differences. First of all, **"." can only repeat one change**. As seen in the example above, **"@a" can do several changes**, and move around as well. Secondly, **"." can only remember the last change.** Executing a register allows you to make any changes and then still use "@a" to replay the

recorded commands. Finally, you can use **26 different registers**. Thus you can remember 26 different command sequences to execute.

94. USING REGISTERS

The registers used for recording are the same ones you used for **yank** and **delete commands**. This allows you to **mix recording with other commands** to manipulate the registers.

Suppose you have recorded a few commands in **register n**. When you execute this with **"@n"** you notice you did something wrong. You could try recording again, but perhaps you will make another mistake. Instead, use this trick:

G	Go to the end of the file.
o<Esc>	Create an empty line.
"np	Put the text from the n register. You now see the commands you typed as text in the file.
{edits}	Change the commands that were wrong. This is just like editing text.
0	Go to the start of the line.
"ny$	Yank the corrected commands into the n register.
dd	Delete the scratch line.

95. APPENDING TO A REGISTER

So far we have used a lowercase letter for the register name. To append to a register, use an **uppercase letter**.

Suppose you have recorded a command to change a word to register c. It works properly, but you would like to add a search for the next word to change. This can be done with:

```
qC/word<Enter>q
```

You start with **"qC"**, which records to the **c register and appends**. Thus writing to an **uppercase register** name means to append to the register with the same letter, but lowercase.

This works both with recording and with yank and delete commands. For example, you want to collect a sequence of lines into the a register. Yank the first line with:

```
"aY
```

Now move to the second line, and type:

```
"AY
```

Repeat this command for all lines. The a register now contains all those lines, in the order you yanked them.

96. Substitution

The "**:substitute**" command enables you to perform string replacements on a whole range of lines. The general form of this command is as follows:

:[range]substitute/from/to/[flags]\

This command changes the "from" string to the "to" string in the lines specified with [range]. For example, you can change "Professor" to "Teacher" in all lines with the following command:

```
:%substitute/Professor/Teacher/
```

The "%" before the command specifies the command works on all lines. Without a range, ":s" only works on the current line.

By default, the ":substitute" command changes only the **first occurrence on each line**. For example, the preceding command changes the line:

    Professor Smith criticized Professor Johnson today.

to:

    Teacher Smith criticized Professor Johnson today.

To **change every occurrence on the line**, you need to add the **g (global)** flag.
The command:

```
:%s/Professor/Teacher/g
```

```
:%s/Professor/Teacher/c
```

Vim finds the first occurrence of "Professor" and displays the text it is about to change. You get the following prompt:

```
replace with Teacher (y/n/a/q/l/^E/^Y)?
```

At this point, you must enter one of the following answers:

y	Yes; make this change.
n	No; skip this match.
a	All; make this change and all remaining ones without further confirmation.

q	Quit; don't make any more changes.
l	Last; make this change and then quit.
CTRL-E	Scroll the text one line up.
CTRL-Y	Scroll the text one line down.

The "from" part of the substitute command is actually a pattern. The same kind as used for the search command. For example, this command only substitutes "the" when it appears at the start of a line:

:s/^the/these/

If you are substituting with a "from" or "to" part that includes a slash, you need to put a backslash before it. A simpler way is to **use another character instead of the slash.** A plus, for example:

:s+one/two+one and two+

97. Command ranges

The **":substitute"** command, and many other **: commands**, can be applied to a selection of lines. This is called a **range**.

- **:1,5s/this/that/g**
  Executes the substitute command on the lines 1 to 5. Line 5 is included.
- **:54s/President/Fool/**
  A single number can be used to address one specific line.
- **:.write otherfile**
  Some commands work on the whole file when you do not specify a range. To make them **work on the current line the "." address is used.**
- **:.,$s/yes/no/**
  To substitute in the lines from the cursor to the end.
  The first line always has number one. How about the last line? **The "$" character is used for this.**
- The **"%"** range that we used before, is actually a short way to say **"1,$"**, from the first to the last line.

98. USING A PATTERN IN A RANGE

Suppose you are editing a chapter in a book, and want to replace all occurrences of "grey" with "gray". But only in this chapter, not in the next one. You know that only chapter boundaries have the word "Chapter" in the first column. This command will work then:

```
:?^Chapter?,/^Chapter/s=grey=gray=g
```

You can see a search pattern is used twice. The first **"?^Chapter?"** finds the line above the current position that matches this pattern. Thus the **?pattern?** range is used to **search backwards**. Similarly, **"/^Chapter/"** is used to **search forward** for the start of the next chapter.

To avoid confusion with the slashes, the "=" character was used in the substitute command here. A slash or another character would have worked as well.

99. ADD AND SUBTRACT

There is a slight error in the above command: If the title of the next chapter had included "grey" it would be replaced as well. Maybe that's what you wanted, but what if you didn't? Then you can specify an offset.

- To search for a pattern and then use the line above it:
  ```
  /Chapter/-1
  ```

- You can use any number instead of the 1. To address the second line below the match:
  ```
  /Chapter/+2
  ```

- The offsets can also be used with the other items in a range. Look at this one:
  ```
  :.+3,$-5
  ```
  This specifies the range that starts three lines below the cursor and ends five lines before the last line in the file.

100. USING MARKS

Instead of figuring out the line numbers of certain positions, remembering them and typing them in a range, you can use marks.

Place the marks as mentioned in chapter 3. For example, use "mt" to mark the top of an area and "mb" to mark the bottom. Then you can use this range to specify the lines between the marks (including the lines with the marks):

```
:'t,'b
```

101. VISUAL MODE AND RANGES

You can select text with Visual mode. If you then press ":" to start a colon command, you will see this:

```
:'<,'>
```

Now you can type the command and it will be applied to the range of lines that was visually selected.

Note:

When using Visual mode to select part of a line, or using CTRL-V to select a block of text, the colon commands will still apply to whole lines. This might change in a future version of Vim.

- The **'<** and **'>** are actually marks, placed at the start and end of the Visual selection. The marks **remain at their position until another Visual selection is made.** Thus you can use the "'<" command to jump to position where the Visual area started. And you can mix the marks with other items:

  ```
  :'>,$
  ```

102. A NUMBER OF LINES

When you know how many lines you want to change, you can type the number and then ":". For example, when you type "**5:**", you will get:

```
:.,.+4
```

Now you can type the command you want to use. It will use the range "." (current line) until ".+4" (four lines down). Thus it spans five lines.

103. The global command

- The ":global" command is one of the more powerful features of Vim. It allows you to find a match for a pattern and execute a command there. The general form is:

  ```
  :[range]global/{pattern}/{command}
  ```

  This is similar to the ":substitute" command. But, instead of replacing the matched text with other text, the command **{command} is executed.**

- Suppose you want to change "foobar" to "barfoo", but only in C++ style comments. These comments start with "//". Use this command:

  ```
  :g+//+s/foobar/barfoo/g
  ```

  This starts with ":g". That is short for ":global", just like ":s" is short for ":substitute". Then the pattern, enclosed in plus characters. Since the pattern we are looking for contains a slash, this uses the plus character to separate the pattern. Next comes the substitute command that changes "foobar" into "barfoo".

The default range for the global command is the whole file. Thus no range was specified in this example. This is different from ":substitute", which works on one line without a range.

The command isn't perfect, since **it also matches lines where "//" appears halfway a line**, and the substitution will also take place before the "//". Just like with ":substitute", any pattern can be used. When you learn more complicated patterns later, you can use them here.

Visual block mode

There is something special about using the "$" command in Visual block mode. When the last motion command used was "$", all lines in the Visual selection will extend until the end of the line, also when the line with the cursor is shorter. This remains effective until you use a motion command that moves the cursor horizontally. Thus using "j" keeps it, "h" stops it.

104. INSERTING TEXT

The command **"I{string}<Esc>"** inserts the text {string} in each line, just left of the visual block. You start by pressing **CTRL-V** to enter visual block mode. Now you move the cursor to define your block. Next you type **I** to enter Insert mode, followed by the text to insert. As you type, the text appears on the first line only.

After you press <Esc> to end the insert, the text will magically be inserted in the rest of the lines contained in the visual selection.

- If the string you insert contains a **newline**, the "I" acts just like a **Normal insert** command and affects **only the first line** of the block.
- The "A" command works the same way, except that it appends **after the right side** of the block. There is one special case for "A": Select a Visual block and then **use "$" to make the block extend to the end of each line**. **Using "A"** now will append the text to the end of each line.

105. CHANGING TEXT

- The Visual block "c" command deletes the block and then throws you into Insert mode to enable you to type in a string. The string will be inserted in each line in the block.

- The "C" command deletes text from the left edge of the block to the end of line. It then puts you in Insert mode so that you can type in a string, which is added to the end of each line.
- Other commands that change the characters in the block:

~	swap case	(a -> A and A -> a)
U	make uppercase	(a -> A and A -> A)
u	make lowercase	(a -> a and A -> a)

106. FILLING WITH A CHARACTER

To fill the whole block with one character, use the **"r"** command.

107. SHIFTING

- The command **">"** shifts the selected text to the right one shift amount, inserting whitespace. The starting point for this shift is the left edge of the visual block.
  The shift amount is specified with the 'shiftwidth' option. To change it to use 4 spaces:
  :set shiftwidth=4

- The "<" command removes one shift amount of whitespace at the left edge of the block. This command is limited by the amount of text that is there; so if there is less than a shift amount of whitespace available, it removes what it can.

108. JOINING LINES

- The **"J"** command joins all selected lines together into one line. Thus it removes the **line breaks**. Actually, the line break, leading white space and trailing white space is replaced by one space. Two spaces are used after a line ending (that can be changed with the 'joinspaces' option).

- The "J" command doesn't require a blockwise selection. It works with "v" and "V" selection in exactly the same way.
  If you don't want the white space to be changed, use the **"gJ"** command.

109. Reading and writing part of a file

When you are writing an e-mail message, you may want to include another file. This can be done with the "**:read {filename}**" command. The text of the file is put below the cursor line.

The ":read" command accepts a **range**. The file will be put below the last line number of this range. Thus "**:$r patch**" appends the file "patch" at the end of the file.

What if you want to read the file above the first line? This can be done with the line number zero. This line doesn't really exist, you will get an error message when using it with most commands. But this command is allowed:

:0read patch

110. WRITING A RANGE OF LINES

Without a **range** it writes the whole file. With a range only the specified lines are written:

- **:.,$write tempo**
- **:.,$write! tempo**

If this file already exists you will get an error message. Vim protects you from accidentally overwriting an existing file. If you know what you are doing and want to **overwrite** the file, append **!**.

111. APPENDING TO A FILE

:.write >>collection

The **">>"** tells Vim the "collection" file is not to be written as a new file, but the line must be appended at the end. You can repeat this as many times as you like.

Formatting text

- **:set textwidth=72**
- **gq**

gqap: To tell Vim to format the current paragraph.

gggqG: to format the whole file.

112. Changing case

operator:

- **gU**
- **gu**
- **g~**
- All these are **operators**, thus they work with any motion command, with text objects and in Visual mode.
- To make an operator work on lines you **double it**. The delete operator is "d", thus to delete a line you use "dd". Similarly, "gugu" makes a whole line lowercase. This can be shortened to **"guu"**. **"gUgU"** is shortened to **"gUU"** and **"g~g~"** to **"g~~"**.

113. Using an external program

!{motion}{program}

!5Gsort<Enter>

- The **"!!"** command filters the current line through a filter. In Unix the "date" command prints the current time and date. **"!!date<Enter>"** replaces the current line with the output of "date". This is useful to add a timestamp to a file.

114. WHEN IT DOESN'T WORK

Starting a shell, sending it text and capturing the output requires that Vim knows how the shell works exactly. When you have problems with filtering, check the values of these options:

'shell'	specifies the program that Vim uses to execute external programs.
'shellcmdflag'	argument to pass a command to the shell
'shellquote'	quote to be used around the command
'shellxquote'	quote to be used around the command and redirection
'shelltype'	kind of shell (only for the Amiga)
'shellslash'	use forward slashes in the command (only for MS-Windows and alikes)
'shellredir'	string used to write the command output into a file

On Unix this is hardly ever a problem, because there are two kinds of shells: "sh" like and "csh" like. Vim checks the 'shell' option and sets related options automatically, depending on whether it sees "csh" somewhere in 'shell'.

On MS-Windows, however, there are many different shells and you might have to tune the options to make filtering work. Check the help for the options for more information.

115. READING COMMAND OUTPUT

To read the contents of the current directory into the file, use this:

on Unix:

:read !ls

on MS-Windows:

:read !dir

The output of the "ls" or "dir" command is captured and inserted in the text, below the cursor. This is similar to reading a file, except that the "!" is used to tell Vim that a command follows.

The command may have arguments. And a range can be used to tell where Vim should put the lines:

:0read !date -u

116. WRITING TEXT TO A COMMAND

The Unix command "wc" counts words. To count the words in the current file:

:write !wc

This is the same write command as before, but instead of a file name the **"!"** character is used and the name of an external command. The written text will be passed to the specified command as its standard input.

117. REDRAWING THE SCREEN

If the external command produced an error message, the display may have been messed up. Vim is very efficient and only redraws those parts of the screen that it knows need redrawing. But it can't know about what another program has written. To tell Vim to redraw the screen:

CTRL-L

Recovering from a crash

Did your computer crash? And you just spent hours editing? Don't panic! Vim stores enough information to be able to restore most of your work. This chapter shows you how to get your work back and explains how the swap file is used.

118. Basic recovery

- **vim -r help.txt**
  Vim will read the **swap file** (used to store text you were editing) and may read bits and pieces of the original file.
- **vim -r ""**

You were editing without a file name, give an empty string as argument.

119. Where is the swap file?

- **vim -r**
  Vim will **list the swap files** that it can find. It will also look in other directories where the swap file for files in the current directory may be located. It will not find swap files in any other directories though, it doesn't search the directory tree.

120. USING A SPECIFIC SWAP FILE

vim -r .help.txt.swo
This is also handy when the swap file is in another directory than expected. Vim recognizes files with the **pattern** *.s[uvw][a-z] as swap files.

If this still does not work, see what file names Vim reports and rename th files accordingly. Check the **'directory'** option to see where Vim may have put the swap file.

Note:

Vim tries to find the swap file by searching the directories in the 'dir' option, looking for files that match "filename.sw?". If wildcard expansion doesn't work (e.g., when the 'shell' option is invalid), Vim does a desperate try to find the file "filename.swp". If that fails too, you will have to give the name of the swapfile itself to be able to recover the file.

Clever tricks

By combining several commands you can make Vim do nearly everything. In this chapter a number of useful combinations will be presented. This uses the commands introduced in the previous chapters and a few more.

121. Replace a word

- **:%s/four/4/g**
- **:%s/\<four/4/g**
- **:%s/\<four\>/4/g**
- **:%s/\<four\>/4/gc**

122. REPLACING IN SEVERAL FILES

Suppose you want to replace a word in more than one file. You could edit each file and type the command manually. It's a lot faster to use record and playback.

Let's assume you have a directory with C++ files, all ending in ".cpp". There is a function called "GetResp" that you want to rename to "GetAnswer".

vim *.cpp Start Vim, defining the argument list to contain all the C++ files. You are now in the first file.

qq Start recording into the q register

:%s/\<GetResp\>/GetAnswer/g

Do the replacements in the first file.

:wnext Write this file and move to the next one.

q Stop recording.

@q Execute the q register. This will replay the substitution and ":wnext". You can verify that this doesn't produce an error message.

999@q Execute the q register on the remaining files.

At the last file you will get an error message, because ":wnext" cannot move to the next file. This stops the execution, and everything is done.

Note:

When playing back a recorded sequence, an error stops the execution. Therefore, make sure you don't get an error message when recording.

There is one catch: If one of the .cpp files does not contain the word "GetResp", you will get an error and replacing will stop. To avoid this, add the "e" flag to the substitute command:

:%s/\<GetResp\>/GetAnswer/ge

The **"e"** flag tells ":substitute" that not finding a match is not an error.

123. Change "Last, First" to "First Last"

You have a list of names in this form:

Doe, John
Smith, Peter

You want to change that to:

John Doe

Peter Smith

This can be done with just one command:

:%s/\([^,]*\), \(.*\)/\2 \1/

This is what the "from" pattern contains:

\([^,]*\), \(.*\)

The first part between \(\) matches "Last" **\(\)**

match anything but a comma **[^,]**

any number of times *****

matches ", " literally **,**

The second part between \(\) matches "First" **\(\)**

any character **.**

any number of times *****

In the "to" part we have **"\2"** and **"\1"**. These are called **backreferences.** They refer to the text matched by the **"\(\)"** parts in the pattern. "\2" refers to the text matched by the second "\(\)", which is the "First" name. "\1" refers to the first "\(\)", which is the "Last" name.

You can use up to nine backreferences in the "to" part of a substitute command. **"\0"** stands for the whole matched pattern. There are a few more special items in a substitute command, see |sub-replace-special|.

magic	nomagic	action	
&	\&	replaced with the whole matched pattern	*s/\&*
\&	&	replaced with &	
\0		replaced with the whole matched pattern	*\0* *s/\0*
\1		replaced with the matched pattern in the first pair of ()	*s/\1*
\2		replaced with the matched pattern in the second pair of ()	*s/\2*
..	..		*s/\3*
\9		replaced with the matched pattern in the ninth	

pair of () *s/\9*

~ \~ replaced with the {string} of the previous

substitute *s~*

\~ ~ replaced with ~ *s/\~*

\u next character made uppercase *s/\u*

\U following characters made uppercase, until \E *s/\U*

\l next character made lowercase *s/\l*

\L following characters made lowercase, until \E *s/\L*

\e end of \u, \U, \l and \L (NOTE: not <Esc>!) *s/\e*

\E end of \u, \U, \l and \L *s/\E*

<CR> split line in two at this point

(Type the <CR> as CTRL-V <Enter>) *s<CR>*

\r idem *s/\r*

\<CR> insert a carriage-return (CTRL-M)

(Type the <CR> as CTRL-V <Enter>) *s/\<CR>*

\n insert a <NL> (<NUL> in the file)

(does NOT break the line) *s/\n*

\b insert a <BS> *s/\b*

\t insert a <Tab> *s/\t*

\\ insert a single backslash *s/\\*

\x where x is any character not mentioned above:

Reserved for future expansion

Examples:

```
 :s/a\|b/xxx\0xxx/g            modifies "a b"      to "xxxaxxx xxxbxxx"
 :s/\([abc]\)\([efg]\)/\2\1/g   modifies "af fa bg" to "fa fa gb"
 :s/abcde/abc^Mde/             modifies "abcde"    to "abc", "de" (two lines)
 :s/$/\^M/                     modifies "abcde"    to "abcde^M"
 :s/\w\+/\u\0/g                modifies "bla bla"  to "Bla Bla"
```

124. Sort a list

125. Reverse line order

The **|:global|** command can be combined with the **|:move|** command to move all the lines before the first line, resulting in a reversed file. The command is:

:global/^/m 0

Abbreviated:

:g/^/m 0

The **"^"** regular expression matches the beginning of the line (even if the line is blank). The **|:move|** command moves the matching line to after the mythical zeroth line, so the current matching line becomes the first line of the file. As the **|:global|** command is not confused by the changing line numbering, **|:global|** proceeds to match all remaining lines of the file and puts each as the first.

This also **works on a range of lines**. First move to above the first line and mark it with "mt". Then move the cursor to the last line in the range and type:

:'t+1,.g/^/m 't

126. Count words

When the whole file is what you want to count the words in, use this command:

g CTRL-G

127. Find a man page

While editing a shell script or C program, you are using a command or function that you want to find the man page for (this is on Unix). Let's first use a simple way: Move the cursor e word to thyou want to find help on and press

K

To find a man page in a specific section, put the section number first. For example, to look in section 3 for "echo":

3K

128. Trim blanks

- Some people find spaces and tabs at the end of a line useless, wasteful, and ugly. To remove whitespace at the end of every line, execute the following command:

  :%s/\s\+$//

  This finds **white space characters (\s)**, 1 or more of them (**\+**), before the end-of-line (**$**).

- Another wasteful use of spaces is placing them before a tab. Often these can be deleted without changing the amount of white space. But not always! Therefore, you can best do this manually. Use this search command:

  /

  You cannot see it, but there is a space before a tab in this command. Thus it's "/<Space><Tab>". Now use "x" to delete the space and check that the amount of white space doesn't change. You might have to insert a tab if it does change. Type "n" to find the next match. Repeat this until no more matches can be found.

129. Find where a word is used

vim `grep -l frame_counter *.c`

130. FINDING EACH LINE

The above command only finds the files in which the word is found. You still have to find the word within the files.

Vim has a built-in command that you can use to search a set of files for a given string. If you want to find all occurrences of "error_string" in all C program files, for example, enter the following command:

:grep error_string *.c

This causes Vim to search for the string "error_string" in all the specified files (*.c). The editor will now open the first file where a match is found and position the cursor on the first matching line. To go to the nextmatching line (no matter in what file it is), use the ":cnext" command. To goto the previous match, use the ":cprev" command. Use ":clist" to see all

the matches and where they are.
The ":grep" command uses the external commands grep (on Unix) or findstr (on Windows). You can change this by setting the option '**grepprg**'.

Editing Effectively

Typing command-line commands quickly

131. Command line abbreviations

This means that the shortest form of ":substitute" is ":s". The following characters are optional. Thus ":su" and ":sub" also work.

132. Command line completion

:edit b<Tab>

Use **CTRL-P** to go through the list in the other direction.

133. LIST MATCHES

When there are many matches, you would like to see an overview. Do this by pressing **CTRL-D**. For example, pressing CTRL-D after:

```
	:set is
```

results in:

```
	:set is
	incsearch  isfname    isident    iskeyword  isprint
	:set is
```

134. THERE IS MORE

The **CTRL-L** command completes the word to the longest unambiguous string. If you type ":edit i" and there are files "info.txt" and "info_backup.txt" you will get ":edit info".

135. Command line history

There are actually four histories. The ones we will mention here are for ":" commands and for "/" and "?" search commands. The "/" and "?" commands share the same history, because they are both search commands. The two other histories are for expressions and input lines for the input() function.

To see all the lines in the history:

```
:history
```

That's the history of ":" commands. The search history is displayed with this command:

```
:history /
```

136. Command line window

Typing the text in the command line works different from typing text in Insert mode. It doesn't allow many commands to change the text. For most commands that's OK, but sometimes you have to type a complicated command. That's where the command line window is useful.

Open the command line window with this command:

```
q:
```

Vim now opens a (small) window at the bottom. It contains the command line history, and an empty line at the end:

Go away and come back

This chapter goes into mixing the use of other programs with Vim. Either by executing program from inside Vim or by leaving Vim and coming back later. Furthermore, this is about the ways to remember the state of Vim and restore it later.

137. Suspend and resume

In case pressing **CTRL-Z** doesn't work, you can also use **":suspend"**. Don't forget to bring Vim back to the foreground, you would lose any changes that you made!

138. Executing shell commands

To execute a single shell command from Vim use **":!{command}"**.

Executing a whole row of programs this way is possible. But a shell is much better at it. You can start a new shell this way:

```
:shell
```

This is similar to using CTRL-Z to suspend Vim. The difference is that a new shell is started.

139. Remembering information: viminfo

Each time you exit Vim it will store this information in a file, the **viminfo** file. When Vim starts again, the viminfo file is read and the information restored.

The **'viminfo'** option is set by default to restore a limited number of items. You might want to set it to remember more information. This is done through the following command:

```
:set viminfo=string
```

The string specifies what to save. The **syntax** of this string is an option character followed by an argument. The **option/argument** pairs are separated by commas.

```
:set viminfo='1000
:set viminfo='1000,f1
:set viminfo='1000,f1,<500
```

Other options you might want to use:

'	specify how many files for which you save marks (a-z)
:	number of lines to save from the command line history
@	number of lines to save from the input line history
/	number of lines to save from the search history
<	controls how many lines are saved for each of the registers. By default, all the lines are saved. If 0, nothing is saved.
r	removable media, for which no marks will be stored (can be used several times)
f	controls whether global marks (A-Z and 0-9) are stored. If this option is 0, none are stored. If it is 1 or you do not specify an f option, the marks are stored.
!	global variables that start with an uppercase letter and

don't contain lowercase letters

h	disable 'hlsearch' highlighting when starting
%	the buffer list (only restored when starting Vim without file arguments)
c	convert the text using 'encoding'
n	name used for the viminfo file (must be the last option)

140. GETTING BACK TO WHERE YOU STOPPED VIM

'0

Vim creates a mark each time you exit Vim. **The last one is '0**. The position that '0 pointed to is made '1. And '1 is made to '2, and so forth. Mark '9 is lost.

141. GETTING BACK TO SOME FILE

You can see a list of files by typing the command:

```
:oldfiles
```

Now you would like to edit the second file, which is in the list preceded by "2:". You type:

```
:e #<2
```

The **"#<2"** item works in the same place as "%" (current file name) and "#" (alternate file name). So you can also split the window to edit the third file:

```
:split #<3
```

142. MOVE INFO FROM ONE VIM TO ANOTHER

You can use the **":wviminfo"** and **":rviminfo"** commands to save and restore the information while still running Vim. This is useful for exchanging register contents between two instances of Vim, for example. In the first Vim do:

```
:wviminfo! ~/tmp/viminfo
```

And in the second Vim do:

```
:rviminfo! ~/tmp/viminfo
```

- The **!** character is used by **":wviminfo"** to forcefully overwrite an existing file. When it is omitted, and the file exists, the information is merged into the file.
- The **!** character used for **":rviminfo"** means that all the information is used, this may overwrite existing information. Without the ! only information that wasn't set is used.
- These commands can also be used to store info and use it again later. You could make a directory full of viminfo files, each containing info for a different purpose.

143. Sessions

Suppose you are editing along, and it is the end of the day. You want to quit work and pick up where you left off the next day. You can do this by saving your editing session and restoring it the next day.

A Vim session contains all the information about what you are editing. This includes things such as the file list, window layout, global variables, options and other information. (Exactly what is remembered is controlled by the 'sessionoptions' option, described below.)

The following command creates a session file:

```
:mksession vimbook.vim
```

Later if you want to restore this session, you can use this command:

```
:source vimbook.vim
```

- If you want to start Vim and restore a specific session, you can use the following command:

  ```
  vim -S vimbook.vim
  ```

  This tells Vim to read a specific file on startup. The 'S' stands forsession (actually, you can source any Vim script with -S, thus it might as well stand for "source").

- What exactly is restored depends on the 'sessionoptions' option. The default value is **"blank,buffers,curdir,folds,help,options,winsize"**.

blank	keep empty windows
buffers	all buffers, not only the ones in a window
curdir	the current directory
folds	folds, also manually created ones
help	the help window
options	all options and mappings
winsize	window sizes

Change this to your liking. To also restore the size of the Vim window, for example, use:

```
:set sessionoptions+=resize
```

144. SESSION HERE, SESSION THERE

The obvious way to use sessions is when working on different projects. Suppose you store you session files in the directory "~/.vim". You are currently working on the "secret" project and have to switch to the "boring" project:

```
:wall
:mksession! ~/.vim/secret.vim
:source ~/.vim/boring.vim
```

145. SESSIONS AND VIMINFO

Some people have to do work on MS-Windows systems one day and on Unix another day. If you are one of them, consider adding "slash" and "unix" to 'sessionoptions'. The session files will then be written in a format that can be used on both systems. This is the command to put in your vimrc file:

```
:set sessionoptions+=unix,slash
```

Vim will use the Unix format then, because the MS-Windows Vim can read and write Unix files, but Unix Vim can't read MS-Windows format session files. Similarly, MS-Windows Vim understands file names with / to separate names, but Unix Vim doesn't understand \.

146. SESSIONS AND VIMINFO

Sessions store many things, but not the position of marks, contents of registers and the command line history. You need to use the viminfo feature for these things.

In most situations you will want to use sessions separately from viminfo. This can be used to switch to another session, but keep the command line history. And yank text into registers in one session, and paste it back in another session.

You might prefer to keep the info with the session. You will have to do this yourself then. Example:

```
:mksession! ~/.vim/secret.vim
:wviminfo! ~/.vim/secret.viminfo
```

And to restore this again:

```
:source ~/.vim/secret.vim
:rviminfo! ~/.vim/secret.viminfo
```

147. Views

A session stores the looks of the whole of Vim. When you want to store the properties for **one window only**, use a **view**. The use of a view is for when you want to edit a file in a specific way.

```
:mkview
```

When you later edit the same file you get the view back with this command:

```
:loadview
```

```
:mkview 1
```

Obviously, you can get this back with:

```
:loadview 1
```

Now you can switch between the two views on the file by using ":loadview" with and without the "1" argument.

You can store up to **ten views for the same file** this way, one unnumbered and nine numbered **1 to 9**.

148. A VIEW WITH A NAME

The second basic way to use views is by storing the view in a file with a name you chose. This view can be loaded while editing another file. Vim will then switch to editing **the file specified in the view.** Thus you can use this to quickly **switch to editing another file**, with all its options set as you saved them.

For example, to save the view of the current file:

```
:mkview ~/.vim/main.vim
```

You can restore it with:

```
:source ~/.vim/main.vim
```

149. Modelines

When editing a specific file, you might set options specifically for that file. Typing these commands each time is boring. Using a session or view for editing a file doesn't work when sharing the file between several people.

The solution for this situation is adding a modeline to the file. This is a line of text that tells Vim the values of options, to be used in this file only.

A typical example is a C program where you make indents by a multiple of 4 spaces. This requires setting the 'shiftwidth' option to 4. This modeline will do that:

```
/* vim:set shiftwidth=4: */
```

The 'modelines' option specifies how many lines at the start and end of the file are inspected for containing a modeline. To inspect ten lines:

```
:set modelines=10
```

The 'modeline' option can be used to switch this off. Do this when you are working as root on Unix or Administrator on MS-Windows, or when you don't trust the files you are editing:

```
:set nomodeline
```

Use this format for the modeline:

```
any-text vim:set {option}={value} ... : any-text
```

Another example:

```
// vim:set textwidth=72 dir=c\:\tmp:  use c:\tmp here
```

There is an extra backslash before the first colon, so that it's included in the ":set" command. The text after the second colon is ignored, thus a remark can be placed there.

Finding the file to edit

Files can be found everywhere. So how do you find them? Vim offers various ways to browse the directory tree. There are commands to jump to a file that is mentioned in another. And Vim remembers which files have been edited before.

150. The file browser

Vim has a **plugin** that makes it possible to edit a directory. Try this:

```
:edit .
:Explore
```

151. The current directory

```
:cd
:lcd
:pwd
```

So long as no ":lcd" command has been used, all windows share the same current directory. Doing a ":cd" command in one window will also change the current directory of the other window.

For a window where ":lcd" has been used a different current directory is remembered. Using ":cd" or ":lcd" in other windows will not change it.

When using a ":cd" command in a window that uses a different current directory, it will go back to using the shared directory.

152. Finding a file

- You are editing a C program that contains this line:

  #include "inits.h"

  You want to see what is in that "inits.h" file. Move the cursor on the name of the file and type:

  gf

  What if the file is not in the current directory? Vim will use the **'path'** option to find the file.
  :set path+=c:/prog/include

- When you know the file name, but it's not to be found in the file, you can type it:

  :find inits.h

  Vim will then use the **'path'** option to try and locate the file. This is the **same as** the **":edit"** command, except for the use of **'path'**.

- To open the found file in a new window use **CTRL-W f** instead of "gf", or use "**:sfind**" instead of ":find".

- A nice way to directly start Vim to edit a file somewhere in the 'path':

  vim "+find stdio.h"

153. The buffer list

The Vim editor uses the **term buffer** to describe a file being edited. Actually, a **buffer** is a copy of the file that you edit. When you finish changing the buffer, you write the contents of the buffer to the file. Buffers not only contain file contents, but also all the **marks**, **settings**, and other stuff that goes with it.

154. HIDDEN BUFFERS

Suppose you are editing the file one.txt and need to edit the file two.txt. You could simply use ":edit two.txt", but since you made changes to one.txt that won't work. You also don't want to write one.txt yet. Vim has a solution for you:

:hide edit two.txt

The buffer "one.txt" disappears from the screen, but Vim still knows that you are editing this buffer, so it keeps the modified text. This is called a hidden buffer: The buffer contains text, but you can't see it.

Be careful! When you have hidden buffers with changes, don't exit Vim without making sure you have saved all the buffers.

155. INACTIVE BUFFERS

156. LISTING BUFFERS

View the buffer list with this command:

```
:buffers
```

A command which does the same, is not so obvious to list buffers, but is much shorter to type:

```
:ls
```

The output could look like this:

```
1 #h  "help.txt"                line 62
2 %a+ "usr_21.txt"              line 1
3     "usr_toc.txt"             line 1
```

The first column contains the buffer number. You can use this to edit the buffer without having to type the name, see below.

After the buffer number come the flags. Then the name of the file and the line number where the cursor was the last time.

The flags that can appear are these (from left to right):

u	Buffer is unlisted |unlisted-buffer|.
%	Current buffer.
#	Alternate buffer.
a	Buffer is loaded and displayed.
h	Buffer is loaded but hidden.
=	Buffer is read-only.
-	Buffer is not modifiable, the 'modifiable' option is off.
+	Buffer has been modified.

157. EDITING A BUFFER

- You can edit a buffer by its number. That avoids having to type the file name:

```
:buffer 2
```

- But the only way to know the number is by looking in the buffer list. You can use the name, or part of it, instead:

:buffer help

Vim will find the best match for the name you type. If there is only one buffer that matches the name, it will be used. In this case "help.txt".

- To open a buffer in a new window:

  :sbuffer 3

158. USING THE BUFFER LIST

You can move around in the buffer list with these commands:

:bnext	go to next buffer
:bprevious	go to previous buffer
:bfirst	go to the first buffer
:blast	go to the last buffer

To remove a buffer from the list, use this command:

:bdelete 3

Note:

Even after removing the buffer with ":bdelete" Vim still remembers it. It's actually made **"unlisted"**, it no longer appears in the list from ":buffers". The **":buffers!"** command will list unlisted buffers (yes, Vim can do the impossible). To really make Vim forget about a buffer, use **":bwipe")**. Also see the '**buflisted**' option.

Editing other files

This chapter is about editing files that are not ordinary files. With Vim you can edit files that are compressed or encrypted. Some files need to be accessed over the internet. With some restrictions, binary files can be edited as well.

159. DOS, Mac and Unix files

Back in the early days, the old Teletype machines used two characters to start a new line. One to move the carriage back to the first position (carriage return, <CR>), another to move the paper up (line feed, <LF>).

When computers came out, storage was expensive. Some people decided that they did not need two characters for end-of-line. The UNIX people decided they could use **<Line Feed>** only for end-of-line. The Apple people standardized on **<CR>**. The MS-DOS (and Microsoft Windows) folks decided to keep the old **<CR><LF>**.

This means that if you try to move a file from one system to another, you have line-break problems. The Vim editor automatically recognizes the different file formats and handles things properly behind your back.

The option 'fileformats' contains the various formats that will be tried when a new file is edited. The following command, for example, tells Vim to try UNIX format first and MS-DOS format second:

```
:set fileformats=unix,dos
```

The three names that Vim uses are:

unix	**<LF>**
dos	**<CR><LF>**
mac	**<CR>**

160. OVERRULING THE FORMAT

If you use the good old Vi and try to edit an MS-DOS format file, you will find that each line ends with a ^M character. (^M is <CR>). The automatic detection avoids this. Suppose you do want to edit the file that way? Then you need to overrule the format:

```
:edit ++ff=unix file.txt
```

The "++" string is an item that tells Vim that an option name follows, which overrules the default for this single command. "++ff" is used for 'fileformat'. You could also use "++ff=mac" or "++ff=dos".

This doesn't work for any option, only **"++ff"** and **"++enc"** are currently implemented. The full names "++fileformat" and "++encoding" also work.

161. CONVERSION

You can use the 'fileformat' option to convert from one file format to another. Suppose, for example, that you have an MS-DOS file named README.TXT that you want to convert to UNIX format. Start by editing the MS-DOS format file:

```
vim README.TXT
```

Vim will recognize this as a dos format file. Now change the file format to UNIX:

:set fileformat=unix
:write

The file is written in Unix format.

162. Files on the internet

There is a much simpler way. Move the cursor to any character of the URL. Then use this command:

gf

With a bit of luck, Vim will figure out which program to use for downloading the file, download it and edit the copy. To open the file in a **new window** use **CTRL-W f**.

163. Encryption

To start editing a new file with encryption, use the **"-x"** argument to start Vim. Example:

vim -x exam.txt

Vim prompts you for a key used for encrypting and decrypting the file:

Enter encryption key:

Vim adds a magic string to the file by which it recognizes that the file was encrypted. If you try to view this file using another program, all you get is garbage. Also, if you edit the file with Vim and enter the wrong key, you get garbage. Vim does not have a mechanism to check if the key is the right one (this makes it much harder to break the key).

164. SWITCHING ENCRYPTION ON AND OFF

To disable the encryption of a file, set the 'key' option to an empty string:

:set key=

Setting the 'key' option to enable encryption is not a good idea, because the password appears in the clear. Anyone shoulder-surfing can read your password.

To avoid this problem, the ":X" command was created. It asks you for an encryption key, just like the "-x" argument did:

:X
Enter encryption key: ******
Enter same key again: ******

165. LIMITS ON ENCRYPTION

The encryption algorithm used by Vim is weak. It is good enough to keep out the casual prowler, but not good enough to keep out a cryptology expert with lots of time on his hands. Also you should be aware that the swap file is not encrypted; so while you are editing, people with superuser privileges can read the unencrypted text from this file.

One way to avoid letting people read your swap file is to avoid using one. If the -n argument is supplied on the command line, no swap file is used (instead, Vim puts everything in memory). For example, to edit the encrypted file "file.txt" without a swap file use the following command:

vim -x -n file.txt

When already editing a file, the swapfile can be disabled with:

:setlocal noswapfile

While the file is in memory, it is in plain text. Anyone with privilege can look in the editor's memory and discover the contents of the file.

If you use a viminfo file, be aware that the contents of text registers are written out in the clear as well.

166. Binary files

- To make sure that Vim does not use its clever tricks in the wrong way, add the "-b argument when starting Vim:

  vim -b datafile
- Many characters in the file will be unprintable. To see them in Hex format:

  :set display=uhex
- Otherwise, the **"ga"** command can be used to see the value of the character under the cursor.
- There might not be many line breaks in the file. To get some overview switch the 'wrap' option off:

  :set nowrap
- To move to a specific byte in the file, use the "go" command. For example, to move to byte 2345:

  2345go

167. USING XXD

First edit the file in binary mode:

vim -b datafile

Now convert the file to a hex dump with xxd:

```
:%!xxd
```

The text will look like this:

```
0000000: 1f8b 0808 39d7 173b 0203 7474 002b 4e49  ....9..;..tt.+NI
0000010: 4b2c 8660 eb9c ecac c462 eb94 345e 2e30  K,.`.....b..4^.0
0000020: 373b 2731 0b22 0ca6 c1a2 d669 1035 39d9  7;'1.".....i.59.
```

You can now view and edit the text as you like. Vim treats the information as ordinary text. Changing the hex does not cause the printable character to be changed, or the other way around.

Finally convert it back with:

```
:%!xxd -r
```

Only changes in the hex part are used. Changes in the printable text part on the right are ignored.

168. Compressed files

You can edit a compressed file just like any other file. These compression methods are currently supported:

.Z	**compress**
.gz	**gzip**
.bz2	**bzip2**

Inserting quickly

When entering text, Vim offers various ways to reduce the number of keystrokes and avoid typing mistakes. Use Insert mode completion to repeat previously typed words. Abbreviate long words to short ones. Type characters that aren't on your keyboard.

169. Making corrections

<BS>

<Del>

CTRL-W

CTRL-U

170. Showing matches

When you type a **)** it would be nice to see with which **(** it matches. To make Vim do that use this command:

:set showmatch

The match will also be shown for **[]** and **{}** pairs.

You can change the time Vim waits with the 'matchtime' option. For example, to make Vim wait one and a half second:

:set matchtime=15

The time is specified in tenths of a second.

171. Completion

Vim can automatically complete words on insertion. You type the first part of a word, press **CTRL-P**, and Vim guesses the rest.

The Vim editor goes through a lot of effort to find words to complete.
By default, it searches the following places:

1. Current file
2. Files in other windows
3. Other loaded files (hidden buffers)
4. Files which are not loaded (inactive buffers)
5. Tag files
6. All files #included by the current file

- You can customize the search order with the 'complete' option.
    - The **'ignorecase'** option is used. When it is set, case differences are ignored when searching for matches.
    - A special option for completion is **'infercase'.** This is useful to find matches while ignoring case ('ignorecase' must be set) but still using the case of the word typed so far. Thus if you type "For" and Vim finds a match "fortunately", it will result in "Fortunately".
- COMPLETING SPECIFIC ITEMS
  If you know what you are looking for, you can use these commands to complete with a certain type of item:

CTRL-X CTRL-F	file names
CTRL-X CTRL-L	whole lines
CTRL-X CTRL-D	macro definitions (also in included files)
CTRL-X CTRL-I	current and included files
CTRL-X CTRL-K	words from a dictionary
CTRL-X CTRL-T	words from a thesaurus
CTRL-X CTRL-]	tags
CTRL-X CTRL-V	Vim command line

If the file name starts with / (Unix) or C:\ (MS-Windows) you can find all files in the file system. For example, type "/u" and CTRL-X CTRL-F. This will match "/usr" (this is on Unix):

172. Repeating an insert

If you press **CTRL-A,** the editor inserts the text you typed the last time you were in Insert mode.

The **CTRL-@** command does a **CTRL-A** and then exits Insert mode. That's a quick way of doing exactly the same insertion again.

173. Copying from another line

The **CTRL-Y** command inserts the character above the cursor. This is useful when you are duplicating a previous line. For example, you have this line of C code:

```
b_array[i]->s_next = a_array[i]->s_next;
```

Now you need to type the same line, but with "s_prev" instead of "s_next". Start the new line, and press CTRL-Y 14 times, until you are at the "n" of "next":

```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_
```

Now you type "prev":

```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev
```

Continue pressing **CTRL-Y** until the following "next":

```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev = a_array[i]->s_
```

Now type "prev;" to finish it off.

The **CTRL-E** command acts like **CTRL-Y** except it inserts the character below the cursor.

174. Inserting a register

The command **CTRL-R {register}** inserts the contents of the register. This is useful to avoid having to type a long word. For example, you need to type this:

```
r = VeryLongFunction(a) + VeryLongFunction(b) + VeryLongFunction(c)
```

The function name is defined in a different file. Edit that file and move the cursor on top of the function name there, and yank it into register v:

```
"vyiw
```

```
r =
```

Now use CTRL-R v to insert the function name:

```
r = VeryLongFunction
```

You continue to type the characters in between the function name, and use **CTRL-R v** two times more.

You could have done the same with completion. Using a register is useful when there are many words that start with the same characters.

If the register contains characters such as <BS> or other special characters, they are interpreted as if they had been typed from the keyboard. If you do not want this to happen (you really want the <BS> to be inserted in the text), use the command **CTRL-R CTRL-R {register}.**

175. Abbreviations

An abbreviation is a short word that takes the place of a long one. For example, **"ad"** stands for **"advertisement".** Vim enables you to type an abbreviation and then will automatically expand it for you.

To tell Vim to expand "ad" into "advertisement" every time you insert it, use the following command:

```
:iabbrev ad advertisement
```

Now, when you type "ad", the whole word "advertisement" will be inserted into the text.

The expansion doesn't happen when typing just "ad". That allows you to type a word like "add", which will not get expanded. Only whole words are checked for abbreviations.

- **ABBREVIATING SEVERAL WORDS**
  It is possible to define an abbreviation that results in multiple words. For example, to define "JB" as "Jack Benny", use the following command:
  :iabbrev JB Jack Benny
  As a programmer, I use two rather unusual abbreviations:
  :iabbrev #b /
  :iabbrev #e <Space>/**
  These are used for creating boxed comments.
  Notice that the #e abbreviation begins with a space. In other words, the first two characters are space-star. Usually Vim **ignores spaces** between the abbreviation and the expansion. To avoid that problem, I spell space as seven characters: <, S, p, a, c, e, >.
  Note:
  ":iabbrev" is a long word to type. ":iab" works just as well. That's abbreviating the abbreviate command!

- **FIXING TYPING MISTAKES**
  :abbreviate teh the
  You can add a whole list of these. Add one each time you discover a common mistake.

- **LISTING ABBREVIATIONS**
  :abbreviate

```
i  #e        ****************************************/
i  #b        /****************************************
i  JB        Jack Benny
i  ad        advertisement
!  teh       the
```

The "i" in the first column indicates Insert mode. These abbreviations are only active in Insert mode. Other possible characters are:

c	Command-line mode	**:cabbrev**
!	both Insert and Command-line mode	**:abbreviate**

Since abbreviations are not often useful in Command-line mode, you will mostly use the ":iabbrev" command. That avoids, for example, that "ad" gets expanded when typing a command like:

```
:edit ad
```

- **DELETING ABBREVIATIONS**
  To get rid of an abbreviation, use the **":unabbreviate"** command. Suppose you have the following abbreviation:
  ```
  :abbreviate @f fresh
  ```
  You can remove it with this command:
  ```
  :unabbreviate @f
  ```
  To remove all the abbreviations:
  ```
  :abclear
  ```

  ":unabbreviate" and **":abclear"** also come in the variants for Insert mode (**":iunabbreviate** and **":iabclear"**) and Command-line mode (**":cunabbreviate"** and **":cabclear"**).

- **REMAPPING ABBREVIATIONS**
  There is one thing to watch out for when defining an abbreviation: The resulting string should not be mapped. For example:
  ```
  :abbreviate @a adder
  :imap dd disk-door
  ```

  When you now type @a, you will get "adisk-doorer". That's not what you want. To avoid this, use the ":noreabbrev" command. It does the same as ":abbreviate", but avoids that the resulting string is used for mappings:
  ```
  :noreabbrev @a adder
  ```
  Fortunately, it's unlikely that the result of an abbreviation is mapped.

176. Entering special characters

- The CTRL-V command is used to insert the next character literally. In other words, any special meaning the character has, it will be ignored. For example:
  ```
  CTRL-V <Esc>
  ```

- You can also use the command **CTRL-V {digits}** to insert a character with the decimal number {digits}. For example, the character number 127 is the <Del> character (but not necessarily the <Del> key!). To insert <Del> type:
  ```
  CTRL-V 127
  ```
  You can enter characters **up to 255** this way. When you type fewer than two digits, a non-digit will terminate the command. To avoid the need of typing a non-digit, prepend

one or two zeros to make three digits.

All the next commands insert a <Tab> and then a dot:

CTRL-V 9.

CTRL-V 09.

CTRL-V 009.

- To enter a character in hexadecimal, use an "x" after the CTRL-V:

  CTRL-V x7f

This also goes up to character 255 (**CTRL-V xff**). You can use **"o"** to type a character as an octal number and two more methods allow you to type up to a 16 bit and a 32 bit number (e.g., for a Unicode character):

CTRL-V o123

CTRL-V u1234

CTRL-V U12345678

177. Digraphs

Some characters are not on the keyboard. For example, the copyright character (©). To type these characters in Vim, you use digraphs, where two characters represent one. To enter a ©, for example, you press three keys:

CTRL-K Co

- To find out what digraphs are available, use the following command:

  :digraphs

  You can exchange the first and second character, if there is no digraph for that combination. Thus **CTRL-K dP** also works. Since there is no digraph for "dP" Vim will also search for a "Pd" digraph.

- You can define your own digraphs. Example:

  :digraph a" ä

  This defines that **CTRL-K a"** inserts an ä character. You can also specify the character with a decimal number. This defines the same digraph:

  :digraph a" 228

178. Normal mode commands

With CTRL-O **{command}** you can **execute any Normal mode command from Insert mode**. For example, to delete from the cursor to the end of the line:

```
CTRL-O D
```

- You can execute only one Normal mode command this way. But you can specify a register or a count. A more complicated example:

  ```
  CTRL-O "g3dw
  ```

  This deletes up to the third word into register g.

Editing formatted text

Text hardly ever comes in one sentence per line. This chapter is about breaking sentences to make them fit on a page and other formatting. Vim also has useful features for editing single-line paragraphs and tables.

179. Breaking lines

自动换行，连续输入不换行，出现空格时会换行：

```
:set textwidth=30

         1         2         3
12345678901234567890123456789012345
I taught programming for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

The Vim editor is not a word processor. In a word processor, if you delete something at the beginning of the paragraph, the line breaks are reworked. In Vim they are not; so if you delete the word "programming" from the first line, all you get is a short line:

```
         1         2         3
12345678901234567890123456789012345
I taught for a
while. One time, I was stopped
```

```
by the Fort Worth police,
because my homework was too
hard. True story.
```

To get the paragraph into shape you use the "**gq**" operator. 实际就是将每行末尾填满。

```
gq
v4jgq
gqap
```

180. Aligning text

相当于居中操作

To center a range of lines, use the following command:

```
:{range}center [width]
```

{range} is the usual command-line range. [width] is an optional line width to use for centering. If [width] is not specified, it defaults to the value of 'textwidth'. (If 'textwidth' is 0, the default is 80.)

For example:

```
:1,5center 40
```

results in the following:

```
  I taught for a while. One
  time, I was stopped by the
Fort Worth police, because my
 homework was too hard. True
           story.
```

- **RIGHT ALIGNMENT**

```
:1,5right 37
```

- **LEFT ALIGNMENT**

```
:{range}left [margin]
```

- **JUSTIFYING TEXT**

Vim has no built-in way of justifying text. However, there is a neat macro package that does the job. To use this package, execute the following command:

```
:runtime macros/justify.vim
```

This Vim script file defines a new visual command "_j". To justify a block of text, highlight the text in Visual mode and then execute "_j".
  Look in the file for more explanations. To go there, do "gf" on this name

```
$VIMRUNTIME/macros/justify.vim.
```

An alternative is to filter the text through an external program. Example:

```
:%!fmt
```

181. Indents and tabs

:set autoindent # 自动缩进

To increase the amount of indent in a line, use the ">" operator. Often this is used as ">>", which adds indent to the current line.

:set shiftwidth=4   #设置使用 >> 命令时缩进的字符数

"**4>>**" will increase the indent of **four lines**.

182. TABSTOP

If you want to make indents a multiple of 4, you set 'shiftwidth' to 4. But when pressing a <Tab> you still get 8 spaces worth of indent. To change this, set the 'softtabstop' option:

```
:set softtabstop=4
```

This will make the <Tab> key insert 4 spaces worth of indent. If there are already four spaces, a <Tab> character is used (saving seven characters in the file). (If you always want spaces and no tab characters, set the **'expandtab'** option.)

Note:

You could set the 'tabstop' option to 4. However, if you edit the
file another time, with 'tabstop' set to the default value of 8, it
will look wrong. In other programs and when printing the indent will
also be wrong. Therefore it is recommended to keep 'tabstop' at eight
all the time. That's the standard value everywhere.

183. CHANGING TABS

You edit a file which was written with a tabstop of 3. In Vim it looks ugly, because it uses the normal tabstop value of 8. You can fix this by setting 'tabstop' to 3. But you have to do

this every time you edit this file.
Vim can change the use of tabstops in your file. First, set 'tabstop' to make the indents look good, then use the ":retab" command:

```
:set tabstop=3
:retab 8
```

The **":retab"** command will change 'tabstop' to 8, while changing the text such that it looks the same. It changes spans of white space into tabs and spaces for this. You can now write the file. Next time you edit it the indents will be right without setting an option.
Warning: When using ":retab" on a program, it may change white space inside a string constant. Therefore it's a good habit to use **"\t**" instead of a real tab.

184. Dealing with long lines

If you switch the 'wrap' option off, each line in the file shows up as one line on the screen. Then the ends of the long lines disappear off the screen to the right.

```
:set guioptions+=b
```

One horizontal scrollbar will appear at the bottom of the Vim window. If you don't have a scrollbar or don't want to use it, use these commands to scroll the text. The cursor will stay in the same place, but it's moved back into the visible text if necessary.

zh	scroll right
4zh	scroll four characters right
zH	scroll half a window width right
ze	scroll right to put the cursor at the end
zl	scroll left
4zl	scroll four characters left
zL	scroll half a window width left
zs	scroll left to put the cursor at the start

- **:set linebreak**

BREAKING AT WORDS

185. MOVING BY VISIBLE LINES

The "j" and "k" commands move to the next and previous lines. When used on a long line, this means moving a lot of screen lines at once.
To **move only one screen line,** use the **"gj"** and **"gk"** commands. When a line doesn't wrap they do the same as "j" and "k". When the line does wrap, they move to a character

displayed one line below or above.
You might like to use these mappings, which bind these movement commands to the cursor keys:

```
:map <Up> gk
:map <Down> gj
```

186. Editing tables

set virtualedit=all

Now you can move the cursor to positions where there isn't any text. This is called **"virtual space"**. Editing a table is a lot easier this way.

Go back to non-virtual cursor movements with:

```
:set virtualedit=
```

- **VIRTUAL REPLACE MODE**

  The disadvantage of using 'virtualedit' is that it "feels" different. You can't recognize tabs or spaces beyond the end of line when moving the cursor around. Another method can be used: Virtual Replace mode.

  gr

  gR

Repeating

An editing task is hardly ever unstructured. A change often needs to be made several times. In this chapter a number of useful ways to repeat a change will be explained.

187. Repeating with Visual mode

Visual mode is very handy for making a change in any sequence of lines. You can see the highlighted text, thus you can check if the correct lines are changed. But making the selection takes some typing. The **"gv"** command selects the same area again. This allows you to do another operation on the same text.

188. Add and subtract

ctrl+A: 在数字上用，加 1, 2ctrl+A 加 2

ctrl+X： 减法

Another example:

006 foo bar
007 foo bar

Using **CTRL-A** on these numbers results in:

007 foo bar
010 foo bar

7 plus one is 10? What happened here is that Vim recognized "007" as an octal number, because there is a leading zero. This notation is often used in C programs. If you do not want a number with leading zeros to be handled as octal, use this:

```
:set nrformats-=octal
```

189. Making a change in many files

Suppose you have a variable called "x_cnt" and you want to change it to "x_counter". This variable is used in several of your C files. You need to change it in all files. This is how you do it.

Put all the relevant files in the argument list:

```
:args *.c
```

This finds all C files and edits the first one. Now you can perform a substitution command on all these files:

```
:argdo %s/\<x_cnt\>/x_counter/ge | update
```

The **":argdo"** command takes an argument that is another command. That command will be executed on all files in the argument list.

The "%s" substitute command that follows works on all lines. It finds the word "x_cnt" with "\<x_cnt\>". The "\<" and "\>" are used to match the whole word only, and not "px_cnt" or "x_cnt2".

The flags for the substitute command include "g" to replace all occurrences of "x_cnt" in the same line. The "e" flag is used to avoid an error message when "x_cnt" does not appear in the file. Otherwise ":argdo" would abort on the first file where "x_cnt" was not found.

The **"|"** separates two commands. The following "update" command writes the file only if it was changed. If no "x_cnt" was changed to "x_counter" nothing happens.

There is also the **":windo"** command, which executes its argument in all windows. And **":bufdo"** executes its argument on all buffers. Be careful with this, because you might have more files in the buffer list than you think. Check this with the ":buffers" command (or ":ls").

190. Using Vim from a shell script

The Vim editor does a superb job as a screen-oriented editor when using Normal mode commands. For batch processing, however, Normal mode commands do not result in clear, commented command files; so here you will use Ex mode instead. This mode gives you a nice command-line interface that makes it easy to put into a batch file. (**"Ex command"** is just another name for a command-line (:) command.)

The Ex mode commands you need are as follows:

```
%s/-person-/Jones/g
write tempfile
quit
```

You put these commands in the file "change.vim". Now to run the editor in batch mode, use this shell script:

```
for file in *.txt; do
  vim -e -s $file < change.vim
  lpr -r tempfile
done
```

The second line runs the Vim editor in **Ex mode (-e argument)** on the file $file and reads commands from the file "change.vim". The **-s** argument tells Vim to operate in silent mode. In other words, do not keep outputting the :prompt, or any other prompt for that matter.

The "lpr -r tempfile" command prints the resulting "tempfile" and deletes it (that's what the -r argument does).

191. READING FROM STDIN

```
ls | vim -
```

If you use the standard input to read text from, you can use the "-S" argument to read a script:

```
producer | vim -S change.vim -
```

192. NORMAL MODE SCRIPTS

If you really want to use Normal mode commands in a script, you can use it like this:

```
vim -s script file.txt ...
```

Note:

"-s" has a different meaning when it is used without "-e". Here it means to source the "script" as Normal mode commands. When used with "-e" it means to be silent, and doesn't use the next argument as a file name.

- The commands in "script" are executed like you typed them. Don't forget that a line break is interpreted as pressing <Enter>. In Normal mode that moves the cursor to the next line.

- To create the script you can edit the script file and type the commands. You need to imagine what the result would be, which can be a bit difficult. Another way is to record the commands while you perform them manually. This is how you do that:

```
vim -w script file.txt ...
```

All typed keys will be written to "script". If you make a small mistake you can just continue and remember to edit the script later.

The **"-w"** argument appends to an existing script. That is good when you want to record the script bit by bit. If you want to start from scratch and start all over, use the **"-W"** argument. It overwrites any existing file.

Search commands and patterns

In chapter 3 a few simple search patterns were mentioned |03.9|. Vim can do much more complex searches. This chapter explains the most often used ones. A detailed specification can be found here: |pattern|

193. Ignoring case

```
:set ignorecase
:set noignorecase
```

Now set the 'smartcase' option:

```
:set ignorecase smartcase
```

With these two options set you find the following matches:

pattern	**matches**
word	word, Word, WORD, WoRd, etc.
Word	Word
WORD	WORD
WoRd	WoRd

194. CASE IN ONE PATTERN

pattern	**matches**
\Cword	word
\CWord	Word
\cword	word, Word, WORD, WoRd, etc.
\cWord	word, Word, WORD, WoRd, etc.

A big advantage of using **"\c"** and **"\C"** is that it sticks with the pattern. Thus if you repeat a pattern from the search history, the same will happen, no matter if 'ignorecase' or 'smartcase' was changed.

Note:

The use of "\" items in search patterns depends on the option.
In this chapter we will assume 'magic' is on, because that is the
standard and recommended setting. If you would change 'magic', many
search patterns would suddenly become invalid.

195. Wrapping around the file end

To turn off search wrapping, use the following command:

```
:set nowrapscan
```

Now when the search hits the end of the file, an error message displays:

```
E385: search hit BOTTOM without match for: forever
```

196. Offsets

For the forward search command "/", the offset is specified by appending a slash (/) and the offset:

/default/2 #在匹配行后再向下走两行

This command searches for the pattern "default" and then moves to the beginning of the second line past the pattern. Using this command on the paragraph above, Vim finds the word "default" in the first line. Then **the cursor is moved two lines down** and lands on "an offset".

If the offset is a simple number, the cursor will be placed at the beginning of the line that many lines from the match. The offset number can be positive or negative. If it is positive, the cursor moves down that many lines; if negative, it moves up.

- **/const/e**
  The "e" offset indicates an offset from the end of the match. It moves the cursor onto the last character of the match.
  /const/e+1
  /const/e-1

- **/const/b+2**
  Moves the cursor to the beginning of the match and then two characters to the right.

- **REPEATING**
  To repeat searching for the previously used search pattern, but with a different offset, leave out the pattern:
  /that
  //e

  Is equal to:
  /that/e

  To repeat with the same offset:
  /
  "n" does the same thing. To repeat while removing a previously used offset:
  //

197. Matching multiple times

- The "*" item specifies that the item before it can match any number of times. Thus:
  /a*

matches "a", "aa", "aaa", etc. But also "" (the empty string), because zero times is included.
The "*" only applies to the item directly before it. Thus "ab*" matches "a", "ab", "abb", "abbb", etc.

- To match a whole string multiple times, it must be grouped into one item. This is done by putting "\(" before it and "\)" after it. Thus this command:

  /\(ab\)*

  Matches: "ab", "abab", "ababab", etc. And also "".

- To avoid matching the empty string, use "\+". This makes the previous item match one or more times.

  /ab\+

  Matches "ab", "abb", "abbb", etc. It does not match "a" when no "b" follows.

- To match an optional item, use "\=". Example:

  /folders\=

  Matches "folder" and "folders".

- To match a specific number of items use the form "\{n,m}". "n" and "m" are numbers. The item before it will be matched "n" to "m" times |inclusive|.
  Example:

  /ab\{3,5}

  matches "abbb", "abbbb" and "abbbbb".

  When "n" is omitted, it defaults to zero. When "m" is omitted it defaults to infinity. When ",m" is omitted, it matches exactly "n" times.
  Examples:

pattern	**match count**
\{,4}	**0, 1, 2, 3 or 4**
\{3,}	**3, 4, 5, etc.**
\{0,1}	**0 or 1, same as \=**
\{0,}	**0 or more, same as ***
\{1,}	**1 or more, same as \+**
\{3}	**3**

- **MATCHING AS LITTLE AS POSSIBLE**
  The items so far match as many characters as they can find. To match as few as possible,

use "\{-n,m}". It works the same as "\{n,m}", except that the minimal amount possible is used.

For example, use:

```
/ab\{-1,3}
```

Will match "ab" in "abbb". Actually, it will never match more than one b, because there is no reason to match more. It requires something else to force it to match more than the lower limit.

The same rules apply to removing "n" and "m". It's even possible to removeboth of the numbers, resulting in "\{-}". This matches the item before it zero or more times, as few as possible. The item by itself always matches zero times. It is useful when combined with something else. Example:

```
/a.\{-}b
```

This matches "axb" in "axbxb". If this pattern would be used:

```
/a.*b
```

It would try to match **as many characters as possible** with ".*", thus it matches "axbxb" as a whole.

198. Alternatives

- The "or" operator in a pattern is **"\|"**. Example:

  ```
  /foo\|bar
  ```

  This matches "foo" or "bar". More alternatives can be concatenated:

  ```
  /one\|two\|three
  ```

  Matches "one", "two" and "three".

- To match multiple times, the whole thing must be placed in "\(" and "\)":

  ```
  /\(foo\|bar\)\+
  ```

  This matches "foo", "foobar", "foofoo", "barfoobar", etc.

- Another example:

  ```
  /end\(if\|while\|for\)
  ```

  This matches "endif", "endwhile" and "endfor".

- A related item is "\&". This requires that both alternatives match in the same place. The resulting match uses the last alternative. Example:

  ```
  /forever\&...
  ```

This matches "for" in "forever". It will not match "fortuin", for example.

199. Character ranges

/[a-z]

/[0-9a-f]

To match the "-" character itself make it the first or last one in the range. These special characters are accepted to make it easier to use them inside a [] range (they can actually be used anywhere in the search pattern):

\e	<Esc>
\t	<Tab>
\r	<CR>
\b	<BS>

There are a few more special cases for [] ranges, see **|/[]|** for the whole story.

- **COMPLEMENTED RANGE**

To avoid matching a specific character, use "^" at the start of the range. The [] item then matches everything but the characters included. Example:

/"[^"]*"

"	**a double quote**
[^"]	**any character that is not a double quote**
*****	**as many as possible**
"	**a double quote again**

This matches "foo" and "3!x", including the double quotes.

- **PREDEFINED RANGES**

A number of ranges are used very often. Vim provides a shortcut for these.
For example:

/\a

Finds alphabetic characters. This is equal to using "/[a-zA-Z]". Here are a few more of these:

item	**matches**	**equivalent**
\d	digit	[0-9]
\D	non-digit	[^0-9]
\x	hex digit	[0-9a-fA-F]
\X	non-hex digit	[^0-9a-fA-F]

\s	white space	[ ]	(<Tab> and <Space>)
\S	non-white characters	[^]	(not <Tab> and <Space>)
\l	lowercase alpha	[a-z]	
\L	non-lowercase alpha	[^a-z]	
\u	uppercase alpha	[A-Z]	
\U	non-uppercase alpha	[^A-Z]	

Note:

Using these predefined ranges works a lot faster than the character range it stands for.
These items can not be used inside []. Thus "[\d\l]" does NOT work to match a digit or lowercase alpha. Use "\(\d\|\l\)" instead.

See **|/\s|** for the whole list of these ranges.

200. Character classes

The character range matches a fixed set of characters. A character class is similar, but with an essential difference: The set of characters can be redefined without changing the search pattern.

For example, search for this pattern:

```
/\f\+
```

The "\f" items stands for file name characters. Thus this matches a sequence of characters that can be a file name.

Which characters can be part of a file name depends on the system you are using. On MS-Windows, the backslash is included, on Unix it is not. This is specified with the 'isfname' option. The default value for Unix is:

```
:set isfname
isfname=@,48-57,/,.,-,_,+,,,#,$,%,~,=
```

For other systems the default value is different. Thus you can make a search pattern with "\f" to match a file name, and it will automatically adjust to the system you are using it on.

Note:

Actually, Unix allows using just about any character in a file name, including white space. Including these characters in 'isfname' would be theoretically correct. But it would make it impossible to find the end of a file name in text. Thus the default value of 'isfname' is a compromise.

The character classes are:

item	**matches**	**option**
\i	identifier characters	'isident'
\I	like \i, excluding digits	
\k	keyword characters	'iskeyword'
\K	like \k, excluding digits	
\p	printable characters	'isprint'
\P	like \p, excluding digits	
\f	file name characters	'isfname'
\F	like \f, excluding digits	

201. Matching a line break

To check for a line break in a specific place, use the "\n" item:

/the\nword

This will match at a line that ends in "the" and the next line starts with "word". To match "the word" as well, you need to match **a space or a line break**. The **item** to use for it is "_s":

/the_sword

To allow any amount of white space:

/the_s\+word

This also matches when "the " is at the end of a line and "   word" at the start of the next one.

"\s" matches white space, **"_s" matches white space or a line break.** Similarly, **"\a"** matches an alphabetic character, and **"_a"** matches an alphabetic character or a line break. The other character classes and ranges can be modified in the same way by inserting a **"_"**.

Many other items can be made to match a line break by prepending "_". For example:
"_." matches any character or a line break.

Note:
"_.*" matches everything until the end of the file. Be careful with

this, it can make a search command very slow.

Another example is "_[]", a character range that includes a line break:

/"_[^"]*"

This finds a text in double quotes that may be split up in several lines.

Folding

Structured text can be separated in sections. And sections in sub-sections. Folding allows you to display a section as one line, providing an overview. This chapter explains the different ways this can be done.

202. Manual folding

- Try it out: Position the cursor in a paragraph and type:

  zfap

  |zf| is an operator and |ap| a text object selection. You can use the |zf| operator with any movement command to **create a fold for the text that it moved ove**r. **|zf|** also works in Visual mode.

- To view the text again, open the fold by typing:

  zo

- And you can close the fold again with:

  zc

- Folds can be nested:

  zr: You could go to each fold and type "zo". To do this faster, use this command.
  zm: This folds M-ore.
  zR: This R-educes folds until there are none left.
  zM: This folds M-ore and M-ore.

  You can quickly disable the folding with the **|zn|** command. Then **|zN|** brings back the folding as it was. **|zi|** toggles between the two. This is a useful way of working.

- It is sometimes difficult to see or remember where a fold is located, thus where a |zo| command would actually work. To see the defined folds:

  :set foldcolumn=4

  This will show a small column on the left of the window to indicate folds. A "+" is shown for a closed fold. A "-" is shown at the start of each open fold and "|" at following lines of the fold.

- To open all folds at the cursor line use **|zO|.**
  To close all folds at the cursor line use **|zC|.**
  To delete a fold at the cursor line use **|zd|.**
  To delete all folds at the cursor line use **|zD|.**

- When in **Insert mode**, the fold at the cursor line is never closed. That allows you to see what you type!
- If you want the line under the cursor always to be open, do this:

  :set foldopen=all
- You can make folds close automatically when you move out of it:

  :set foldclose=all

203. Saving and restoring folds

To save the folds use the |:mkview| command:

:mkview

When you come back to the same file later, you can load the view again:

:loadview

You can store up to ten views on one file. For example, to save the current setup as the third view and load the second view:

:mkview 3
:loadview 2

Note that when you insert or delete lines the views might become invalid. Also check out the **'viewdir'** option, which specifies where the views are stored. You might want to delete old views now and then.

204. Folding by indent

Defining folds with **|zf|** is a lot of work. If your text is structured by giving lower level items a larger indent, you can use the indent folding method. This will create folds for every sequence of lines with the same indent. Lines with a larger indent will become nested folds. This works well with many programming languages.

Try this by setting the **'foldmethod'** option:

```
:set foldmethod=indent
```

Then you can use the **|zm|** and **|zr|** commands to fold more and reduce folding.

When you use the |zr| and |zm| commands you actually increase or decrease the 'foldlevel' option. You could also set it directly:

```
:set foldlevel=3
```

This means that all folds with three times a 'shiftwidth' indent or more will be closed. The lower the foldlevel, the more folds will be closed. When 'foldlevel' is zero, all folds are closed. |zM| does set 'foldlevel' to zero. The opposite command |zR| sets 'foldlevel' to the deepest fold level that is
present in the file.

Thus there are **two ways to open and close the folds:**
(A) By setting the fold level.
  This gives a very quick way of "zooming out" to view the structure of the
  text, move the cursor, and "zoom in" on the text again.

(B) By using |zo| and |zc| commands to open or close specific folds.
  This allows opening only those folds that you want to be open, while other
  folds remain closed.

More about folding by indent in the reference manual: **|fold-indent|**

205. Folding with markers

Markers in the text are used to specify the start and end of a fold region. This gives precise control over which lines are included in a fold. The disadvantage is that the text needs to be modified.

Try it:

```
:set foldmethod=marker
```

Example text, as it could appear in a C program:

```
/* foobar () {{{ */
int foobar()
{
        /* return a value {{{ */
        return 42;
        /* }}} */
}
/* }}} */
```

Example:

```
/* global variables {{{1 */
int varA, varB;

/* functions {{{1 */
/* funcA() {{{2 */
void funcA() {}

/* funcB() {{{2 */
void funcB() {}
/* }}}1 */
```

More about folding with markers in the reference manual: **|fold-marker|**

206. Folding by expression

This is similar to folding by indent, but instead of using the indent of a line **a user function is called to compute the fold level of a line.** You can use this for text where something in the text indicates which lines belong together. An example is an e-mail message where the quoted text is indicated by a ">" before the line. To fold these quotes use this:

```
:set foldmethod=expr
:set foldexpr=strlen(substitute(substitute(getline(v:lnum),'\\s','',\"g\"),'[^>].*','',''))
```

You can try it out on this text:

```
> quoted text he wrote
> quoted text he wrote
> > double quoted text I wrote
```

> > double quoted text I wrote

Explanation for the 'foldexpr' used in the example (inside out):

```
getline(v:lnum)                  gets the current line
substitute(...,'\\s','','g')     removes all white space from the line
substitute(...,'[^>].*','','')   removes everything after leading '>'s
strlen(...)                      counts the length of the string, which
                                 is the number of '>'s found
```

Note that a backslash must be inserted before every space, double quote and backslash for the ":set" command. If this confuses you, do

```
:set foldexpr
```

to check the actual resulting value. To correct a complicated expression, use the command-line completion:

```
:set foldexpr=<Tab>
```

Where <Tab> is a real Tab. Vim will fill in the previous value, which you can then edit.

When the expression gets more complicated you should put it in a function and set 'foldexpr' to call that function.

More about folding by expression in the reference manual: **|fold-expr|**

207. Folding unchanged lines

This is useful when you set the **'diff'** option in the same window. The |vimdiff| command does this for you. Example:

```
:setlocal diff foldmethod=diff scrollbind nowrap foldlevel=1
```

Do this in every window that shows a different version of the same file. You will clearly see the differences between the files, while the text that didn't change is folded.

For more details see **|fold-diff|.**

Moving through programs

The creator of Vim is a computer programmer. It's no surprise that Vim contains many features to aid in writing programs. Jump around to find where identifiers are defined and used. Preview declarations in a separate window.
There is more in the next chapter.

208. Using tags

What is a tag? It is a location where an identifier is defined. An example is a function definition in a C or C++ program. A list of tags is kept in a tags file. This can be used by Vim to directly jump from any place to the tag, the place where an identifier is defined.

To generate the tags file for all C files in the current directory, use the following command:

```
ctags *.c
```

Now when you are in Vim and you want to go to a function definition, you can jump to it by using the following command:

```
:tag startlist
```

This command will find the function "startlist" even if it is in another file.

The **CTRL-]** command jumps to the tag of the word that is under the cursor.

The "**:tags**" command shows the list of tags that you traversed through:

```
:tags
  # TO tag        FROM line  in file/text
  1  1 write_line         8  write_block.c
  2  1 write_char         7  write_line.c
>
```

Now to go back. The **CTRL-T** command goes to the preceding tag. In the example above you get back to the "write_line" function, in the call to "write_char".

This command takes a count argument that indicates how many tags to jump back. You have gone forward, and now back. Let's go forward again. The following command goes to the tag **on top of the list**:

```
:tag
```

You can prefix it with a count and jump forward that many tags. For example: **":3tag"**. CTRL-T also can be preceded with a count.

These commands thus allow you to go down a call tree with **CTRL-]** and back up again with CTRL-T. Use ":tags" to find out where you are.

- **SPLIT WINDOWS**
  :stag tagname

  To split the current window and jump to the tag under the cursor use this command:

  ```
  CTRL-W ]
  ```

  If a count is specified, the new window will be that many lines high.

- **MORE TAGS FILES**
  When you have files in many directories, you can create a tags file in each of them. Vim will then only be able to jump to tags within that directory.

  To find more tags files, set the 'tags' option to include all the relevant tags files. Example:

  ```
  :set tags=./tags,./../tags,./*/tags
  ```

  This finds a tags file in the same directory as the current file, one directory level higher and in all subdirectories.

  This is quite a number of tags files, but it may still not be enough. For example, when editing a file in "~/proj/src", you will not find the tags file "~/proj/sub/tags". For this situation Vim offers to search a whole directory tree for tags files. Example:

  ```
  :set tags=~/proj/**/tags
  ```

- **ONE TAGS FILE**
  When Vim has to search many places for tags files, you can hear the disk rattling. It may get a bit slow. In that case it's better to spend this time while generating one big tags file. You might do this overnight.

  This requires the Exuberant ctags program, mentioned above. It offers an argument to search a whole directory tree:

  ```
  cd ~/proj
  ctags -R .
  ```

  The nice thing about this is that Exuberant ctags recognizes various file types. Thus this doesn't work just for C and C++ programs, also for Eiffel and even Vim scripts. See the ctags documentation to tune this.

Now you only need to tell Vim where your big tags file is:

```
:set tags=~/proj/tags
```

- **MULTIPLE MATCHES**

When a function is defined multiple times (or a method in several classes), the **":tag"** command will jump to the first one. If there is a match in the current file, that one is used first.

You can now jump to other matches for the same tag with:

```
:tnext
```

Repeat this to find further matches. If there are many, you can select which one to jump to:

```
:tselect tagname
```

To move between the matching tags, these commands can be used:

:tfirst	go to first match
:[count]tprevious	go to [count] previous match
:[count]tnext	go to [count] next match
:tlast	go to last match

If [count] is omitted then one is used.

- **GUESSING TAG NAMES**

Sometimes you only know part of the name of a function. Or you have many tags that start with the same string, but end differently. Then you can tell Vim to use a pattern to find the tag.

Suppose you want to jump to a tag that contains "block". First type this:

```
:tag /block
```

Now use command line completion: press <Tab>. Vim will find all tags that contain "block" and use the first match.

The **"/"** before a tag name tells Vim that **what follows is not a literal tag name, but a pattern.** You can use all the items for **search patterns** here. For example, suppose you want to select a tag that starts with "write_":

```
:tselect /^write_
```

- **A TAGS BROWSER**

Since CTRL-] takes you to the definition of the identifier under the cursor, you can use a list of identifier names as a table of contents. Here is an example.
First create a list of identifiers (this requires Exuberant ctags):

```
ctags --c-types=f -f functions *.c
```

Now start Vim without a file, and edit this file in Vim, in a vertically split window:

```
vim
:vsplit functions
```

The window contains a list of all the functions. There is some more stuff, but you can ignore that. Do ":setlocal ts=99" to clean it up a bit.
In this window, define a mapping:

```
:nnoremap <buffer> <CR> 0ye<C-W>w:tag <C-R>"<CR>
```

Move the cursor to the line that contains the function you want to go to.
Now press <Enter>. Vim will go to the other window and jump to the selected function.

- **RELATED ITEMS**

You can set 'ignorecase' to make case in tag names be ignored.

The **'tagbsearch'** option tells if the tags file is sorted or not. The default is to assume a sorted tags file, which makes a tags search a lot faster, but doesn't work if the tags file isn't sorted.

The **'taglength'** option can be used to tell Vim the number of significant characters in a tag. When you use the SNiFF+ program, you can use the Vim interface to it |sniff|. SNiFF+ is a commercial program.

Cscope is a free program. It does not only find places where an identifier is declared, but also where it is used. See |cscope|.

209. The preview window

To open a preview window to display the function "write_char":

```
:ptag write_char
```

Vim will open a window, and jumps to the tag "write_char". Then it takes you back to the

original position. Thus you can continue typing without the need to use a **CTRL-W** command.
If the name of a function appears in the text, you can get its definition in the preview window with:

```
CTRL-W }
```

There is a script that automatically displays the text where the word under the cursor was defined. See **|CursorHold-example|**.

To close the preview window use this command:

```
:pclose
```

To edit a specific file in the preview window, use **":pedit"**. This can be useful to edit a header file, for example:

```
:pedit defs.h
```

Finally, **":psearch"** can be used to find a word in the current file and any included files and display the match in the preview window. This is especially useful when using library functions, for which you do not have a tags file. Example:

```
:psearch popen
```

This will show the "stdio.h" file in the preview window, with the function prototype for popen():

```
FILE    *popen __P((const char *, const char *));
```

You can specify the height of the preview window, when it is opened, with the **'previewheight'** option.

210. Moving through a program

Since a program is structured, Vim can recognize items in it. Specific commands can be used to move around.
C programs often contain constructs like this:

```
#ifdef USE_POPEN
   fd = popen("ls", "r")
#else
   fd = fopen("tmp", "w")
#endif
```

But then much longer, and possibly nested. Position the cursor on the **"#ifdef"** and press **%**. Vim will jump to the **"#else"**. Pressing **%** again takes you to the **"#endif"**. Another % takes you to the "#ifdef" again.

When the construct is nested, Vim will find the matching items. This is a good way to check if you didn't forget an "#endif".

When you are somewhere inside a "#if" - "#endif", you can jump to the start of it with:

[#

If you are not after a "#if" or "#ifdef" Vim will beep. To jump forward to the next "#else" or "#endif" use:

]#

- **MOVING IN CODE BLOCKS**

  In C code blocks are enclosed in {}. These can get pretty long. To move to the start of the outer block use the **"[["** command. Use **"]["** to find the end. This assumes that the "{" and "}" are in the first column.

  The **"[{"** command moves to the start of the current block. It skips over pairs of {} at the same level. **"]}"** jumps to the end.

- **MOVING IN BRACES**

  The **"[("** and **"])"** commands work similar to "[{" and "]}", except that they work on () pairs instead of {} pairs.

- **MOVING IN COMMENTS**

  To move back to the start of a comment use **"[/"**. Move forward to the end of a comment with **"]/"**. This only works for /* - */ comments.

211. Finding global identifiers

You are editing a C program and wonder if a variable is declared as "int" or "unsigned". A quick way to find this is with the "[I" command.

Suppose the cursor is on the word "column". Type:

[I

Vim will list the matching lines it can find. Not only in the current file, but also in **all included files** (and files included in them, etc.).

However, a few things must be right for "[I" to do its work. First of all, the 'include' option must specify how a file is included. The default value works for C and C++. For other languages you will have to change it.

- **LOCATING INCLUDED FILES**
  Vim will find included files in the places specified with the 'path' option. If a directory is missing, some include files will not be found. You can discover this with this command:
  :checkpath
  It will list the include files that could not be found. Also files included by the files that could be found.

- **JUMPING TO A MATCH**
  "[I" produces a list with only one line of text. When you want to have a closer look at the first item, you can jump to that line with the command:
  [<Tab>

  You can also use **"[CTRL-I"**, since CTRL-I is the same as pressing <Tab>.

  The list that "[I" produces has a number at the start of each line. When you want to jump to another item than the first one, type the number first:
  3[<Tab>

  Will jump to the third item in the list. Remember that you can use CTRL-O to jump back to where you started from.

  RELATED COMMANDS

[i	only lists the first match
]I	only lists items below the cursor
]i	only lists the first item below the cursor

- **FINDING DEFINED IDENTIFIERS**
  The **"[I"** command finds any identifier. To find only macros, defined with "#define" use:
  [D

  Again, this searches in included files. The 'define' option specifies what a line looks like that defines the items for "[D". You could change it to make it work with other languages than C or C++.

The commands related to "[D" are:

[d	only lists the first match
]D	only lists items below the cursor
]d	only lists the first item below the cursor

212. Finding local identifiers

The "[I" command searches included files. To search in the current file only, and jump to the first place where the word under the cursor is used:

gD

Hint: **Goto Definition**. This command is very useful to find a variable or function that was declared locally ("static", in C terms). To restrict the search even further, and look only in the current function, use this command:

gd

This will go back to the start of the current function and find the first occurrence of the word under the cursor. Actually, it searches backwards to an empty line above a "{" in the first column. From there it searches forward for the identifier.

Editing programs

Vim has various commands that aid in writing computer programs. Compile a program and directly jump to reported errors. Automatically set the indent for many languages and format comments.

213. Compiling

The following command runs the program "make" (supplying it with any argument you give) and captures the results:

:make {arguments}

The following command goes to where the next error occurs:

:cnext

When there is not enough room, Vim will shorten the error message. To see the whole message use:

:cc

You can get an overview of all the error messages with the **":clist"** command. To see all the messages add a **"!"** to the command.

Vim will highlight the current error. To go back to the previous error, use:

:cprevious

Other commands to move around in the error list:

:cfirst	**to first error**
:clast	**to last error**
:cc 3	**to error nr 3**

214. USING ANOTHER COMPILER

The name of the program to run when the ":make" command is executed is defined by the **'makeprg'** option. Usually this is set to "make", but Visual C++ users should set this to "nmake" by executing the following command:

:set makeprg=nmake

You can also include arguments in this option. Special characters need to be escaped with a backslash. Example:

:set makeprg=nmake\ -f\ project.mak

You can include special Vim keywords in the command specification. The % character expands to the name of the current file. So if you execute the command:

:set makeprg=make\ %

When you are editing main.c, then ":make" executes the following command:

make main.c

This is not too useful, so you will refine the command a little and use the :r (root) modifier:

:set makeprg=make\ %:r.o

Now the command executed is as follows:

make main.o

More about these modifiers here: **|filename-modifiers|.**

215. OLD ERROR LISTS

Suppose you **":make"** a program. There is a warning message in one file and an error message in another. You fix the error and use ":make" again to check if it was really fixed. Now you want to look at the warning message. It doesn't show up in the last error list, since the file with the warning wasn't compiled again. You can go back to the previous error list with:

```
:colder
```

Then use **":clist"** and **":cc {nr}"** to jump to the place with the warning.

To go forward to the next error list:

```
:cnewer
```

Vim remembers **ten error lists.**

216. SWITCHING COMPILERS

You have to tell Vim what format the error messages are that your compiler produces. This is done with the 'errorformat' option. The syntax of this option is quite complicated and it can be made to fit almost any compiler. You can find the explanation here: |errorformat|.

You might be using various different compilers. Setting the 'makeprg' option, and especially the 'errorformat' each time is not easy. Vim offers a simple method for this. For example, to switch to using the Microsoft Visual C++ compiler:

```
:compiler msvc
```

This will find the Vim script for the "msvc" compiler and set the appropriate options.

You can write your own compiler files. See |write-compiler-plugin|.

217. OUTPUT REDIRECTION

The **":make"** command redirects the **output of the executed program** to an **error file.** How this works depends on various things, such as the 'shell'. If your **":make"** command doesn't capture the output, check the **'makeef'** and **'shellpipe'** options. The **'shellquote'** and **'shellxquote'** options might also matter.

In case you can't get ":make" to redirect the file for you, an alternative is to compile the program in another window and redirect the output into a file.

Then have Vim read this file with:

```
:cfile {filename}
```

Jumping to errors will work like with the ":make" command.

218. Indenting C style text

A program is much easier to understand when the lines have been properly indented. Vim offers various ways to make this less work. For C or C style programs like Java or C++, set the **'cindent'** option. Vim knows a lot about C programs and will try very hard to automatically set the indent for you. Set the 'shiftwidth' option to the amount of spaces you want for a deeper level. Four spaces will work fine. One ":set" command will do it:

```
:set cindent shiftwidth=4
```

- When you have code that is badly formatted, or you inserted and deleted lines, you need to re-indent the lines. The "=" operator does this. The simplest form is:

```
==
```

This indents the **current line.** Like with all operators, there are three ways to use it. In Visual mode "=" indents the selected lines. A useful text object is "a{". This selects the current {} block. Thus, to re-indent the code block the cursor is in:

```
=a{
```

I you have really badly indented code, you can re-indent the whole file with:

```
gg=G
```

However, don't do this in files that have been carefully indented manually. The automatic indenting does a good job, but in some situations you might want to overrule it.

- **SETTING INDENT STYLE**

Different people have different styles of indentation. By default Vim does a pretty good job of indenting in a way that 90% of programmers do. There are different styles, however; so if you want to, you can customize the indentation style with the 'cinoptions' option.

By default 'cinoptions' is empty and Vim uses the default style. You can add various items where you want something different. For example, to make curly braces be placed like this:

```
if (flag)
  {
    i = 8;
    j = 0;
    }
```

Use this command:

```
:set cinoptions+={2
```

There are many of these items. See **|cinoptions-values|.**

219. Automatic indenting

You don't want to switch on the 'cindent' option manually every time you edit a C file. This is how you make it work automatically:

```
:filetype indent on
```

If you don't like the automatic indenting, you can switch it off again:

```
:filetype indent off
```

- If you don't like the indenting for one specific type of file, this is how you avoid it. Create a file with just this one line:

```
:let b:did_indent = 1
```

Now you need to write this in a file with a specific name:

```
{directory}/indent/{filetype}.vim
```

The {filetype} is the name of the file type, such as "cpp" or "java". You can see the exact name that Vim detected with this command:

```
:set filetype
```

In this file the output is:

```
filetype=help
```

Thus you would use "help" for {filetype}.

For the **{directory}** part you need to use your runtime directory. Look at the output of this command:

```
set runtimepath
```

Now use the first item, the name before the first comma. Thus if the output looks like this:

```
runtimepath=~/.vim,/usr/local/share/vim/vim60/runtime,~/.vim/after
```

You use "~/.vim" for {directory}. Then the resulting file name is:

~/.vim/indent/help.vim

Instead of switching the indenting off, you could write your own indent file. How to do that is explained here: **|indent-expression|.**

220. Other indenting

The most simple form of automatic indenting is with the **'autoindent'** option. It uses the indent from the previous line. A bit smarter is the **'smartindent'** option.

With **'smartindent'** set, an extra level of indentation is added for each { and removed for each }. An extra level of indentation will also be added for any of the words in the 'cinwords' option. Lines that begin with # are treated specially: all indentation is removed. This is done so that preprocessor directives will all start in column 1. The indentation is restored for the next line.

- **CORRECTING INDENTS**

When you are using 'autoindent' or 'smartindent' to get the indent of the previous line, there will be many times when you need to add or remove one **'shiftwidth'** worth of indent. A quick way to do this is using the **CTRL-D** and **CTRL-T** commands in Insert mode.

For example, you are typing a shell script that is supposed to look like this:

```
if test -n a; then
   echo a
   echo "-------"
fi
```

Start off by setting these options:

```
:set autoindent shiftwidth=3
```

You start by typing the first line, <Enter> and the start of the second line:

```
if test -n a; then
echo
```

Now you see that you need an extra indent. Type **CTRL-T**. The result:

```
if test -n a; then
   echo
```

The **CTRL-T** command, in **Insert mode**, adds one 'shiftwidth' to the indent, no matter where in the line you are.

You continue typing the second line, <Enter> and the third line. This time the indent is OK. Then <Enter> and the last line. Now you have this:

```
if test -n a; then
   echo a
   echo "-------"
   fi
```

To remove the superfluous indent in the last line press CTRL-D. This deletes one 'shiftwidth' worth of indent, no matter where you are in the line.

When you are in Normal mode, you can use the **">>"** and **"<<"** commands to shift lines. **">"** and **"<"** are operators, thus you have the usual three ways to specify the lines you want to indent. A useful combination is:

```
>i{
```

This adds one indent to the current block of lines, inside {}. The { and } lines themselves are left unmodified. **">a{"** includes them.

221. Tabs and spaces

If you are using a combination of tabs and spaces, you just edit normally. The Vim defaults do a fine job of handling things.

You can make life a little easier by setting the 'softtabstop' option. This option tells Vim to make the **<Tab>** key look and feel as if tabs were set at the value of **'softtabstop'**, but actually use a combination of tabs and spaces.

After you execute the following command, every time you press the <Tab> key the cursor moves to the next 4-column boundary:

```
:set softtabstop=4
```

When you start in the first column and press <Tab>, you get 4 spaces inserted in your text. The second time, Vim takes out the 4 spaces and puts in a <Tab> (thus taking you to column 8). Thus Vim uses as many <Tab>s as possible, and then fills up with spaces.

When backspacing it works the other way around. A <BS> will always delete the amount specified with **'softtabstop'**. Then <Tab>s are used as many as possible and spaces to fill the gap.

An alternative is to use the **'smarttab'** option. When it's set, Vim uses 'shiftwidth' for a <Tab> typed in the indent of a line, and a real <Tab> when typed after the first non-blank character. However, <BS> doesn't work like with 'softtabstop'.

- **JUST SPACES**
  If you want absolutely no tabs in your file, you can set the 'expandtab' option:
  ```
  :set expandtab
  ```
  When this option is set, the <Tab> key inserts a series of spaces.

- **CHANGING TABS IN SPACES (AND BACK)**
  Setting **'expandtab'** does not affect any existing tabs. In other words, any tabs in the document remain tabs. If you want to convert tabs to spaces, use the ":retab" command. Use these commands:
  ```
  :set expandtab
  :%retab
  ```

  Now Vim will have changed all indents to use spaces instead of tabs. However, all tabs that come after a non-blank character are kept. If you want these to be converted as well, add a **!**:
  ```
  :%retab!
  ```

  This is a little bit dangerous, because it can also change tabs inside a string. To check if these exist, you could use this:
  ```
  /"[^"\t]*\t[^"]*"
  ```

  It's recommended not to use hard tabs inside a string. Replace them with "\t" to avoid trouble.

  The other way around works just as well:
  ```
  :set noexpandtab
  :%retab!
  ```

222. Formatting comments

One of the great things about Vim is that it understands comments. You can ask Vim to format a comment and it will do the right thing.

Suppose, for example, that you have the following comment:
```
/*
 * This is a test
 * of the text formatting.
 */
```
You then ask Vim to format it by positioning the cursor at the start of the comment and

type:

```
	gq]/
```

"gq" is the operator to format text. "]/" is the motion that takes you to the end of a comment. The result is:

```
	/*
	 * This is a test of the text formatting.
	 */
```

To add a new line to the comment, position the cursor on the middle line and press "o". The result looks like this:

```
	/*
	 * This is a test of the text formatting.
	 *
	 */
```

Vim has automatically inserted a star and a space for you. Now you can type the comment text. When it gets longer than 'textwidth', Vim will break the line. Again, the star is inserted automatically:

```
	/*
	 * This is a test of the text formatting.
	 * Typing a lot of text here will make Vim
	 * break
	 */
```

For this to work some flags must be present in 'formatoptions':

r	insert the star when typing <Enter> in Insert mode
o	insert the star when using "o" or "O" in Normal mode
c	break comment text according to 'textwidth'

See **|fo-table|** for more flags.

223. DEFINING A COMMENT

Many single-line comments start with a specific character. In C++ // is used, in Makefiles #, in Vim scripts ". For example, to make Vim understand C++ comments:

```
	:set comments=://
```

The colon separates the flags of an item from the text by which the comment is recognized.

The general form of an item in 'comments' is:

```
{flags}:{text}
```

The {flags} part can be **empty**, as in this case.

Several of these items can be **concatenated**, **separated by commas**. This allows recognizing different types of comments at the same time. For example, let's edit an e-mail message. When replying, the text that others wrote is preceded with ">" and **"!"** characters. This command would work:

```
:set comments=n:>,n:!
```

There are two items, one for comments starting with ">" and one for comments that start with **"!".** Both use the flag **"n"**. This means that these comments nest. Thus a line starting with ">" may have another comment after the ">". This allows formatting a message like this:

```
> ! Did you see that site?
> ! It looks really great.
> I don't like it.  The
> colors are terrible.
What is the URL of that
site?
```

Try setting 'textwidth' to a different value, e.g., 80, and format the text by Visually selecting it and typing "gq". The result is:

```
> ! Did you see that site?  It looks really great.
> I don't like it.  The colors are terrible.
What is the URL of that site?
```

You will notice that Vim did not move text from one type of comment to another. The "I" in the second line would have fit at the end of the first line, but since that line starts with "> !" and the second line with ">", Vim knows that this is a different kind of comment.

- **A THREE PART COMMENT**

  A C comment starts with "/*", has "*" in the middle and "*/" at the end. The entry in 'comments' for this looks like this:

```
:set comments=s1:/*,mb:*,ex:*/
```

The start is defined with "s1:/*". The "s" indicates the start of a three-piece comment. The colon separates the flags from the text by which the comment is recognized: "/*". There is one flag: "1". This tells Vim that the middle part has an offset of one space.
The middle part "mb:*" starts with "m", which indicates it is a middle part. The "b" flag means that a blank must follow the text. Otherwise Vim would consider text like "*pointer" also to be the middle of a comment.
The end part "ex:*/" has the "e" for identification. The "x" flag has a special meaning. It means that after Vim automatically inserted a star, typing / will remove the extra space.

For more details see |**format-comments|**.

Exploiting the GUI

Vim works well in a terminal, but the GUI has a few extra items. A file browser can be used for commands that use a file. A dialog to make a choice between alternatives. Use keyboard shortcuts to access menu items quickly.

224. The file browser

```
:browse split
```

You can also specify a file name argument. This is used to tell the file browser where to start. Example:

```
:browse split /etc
```

This can be changed with the 'browsedir' option. It can have one of three values:

last	Use the last directory browsed (default)
buffer	Use the same directory as the current buffer
current	use the current directory

For example, when you are in the directory "/usr", editing the file "/usr/local/share/readme", then the command:

```
:set browsedir=buffer
:browse edit
```

Will start the browser in "/usr/local/share". Alternatively:

```
:set browsedir=current
:browse edit
```
Will start the browser in "/usr".

225. Confirmation

Vim protects you from accidentally overwriting a file and other ways to lose changes. If you do something that might be a bad thing to do, Vim produces an error message and suggests appending ! if you really want to do it.

To avoid retyping the command with the !, you can make Vim give you a dialog. You can then press "OK" or "Cancel" to tell Vim what you want.

For example, you are editing a file and made changes to it. You start editing another file with:
```
:confirm edit foo.txt
```
Just like **":browse"**, the **":confirm"** command can be prepended to most commands that edit another file. They can also be combined:
```
:confirm browse edit
```

226. Vim window position and size

To see the current Vim window position on the screen use:
```
:winpos
```
The position is given in screen pixels. Now you can use the numbers to move Vim somewhere else. For example, to move it to the left a hundred pixels:
```
:winpos 172 103
```
You can use this command in your startup script to position the window at a specific position.

The size of the Vim window is computed in characters.
```
:set lines columns
```
To change the size set the 'lines' and/or 'columns' options to a new value:
```
:set lines=50
:set columns=80
```

Obtaining the size works in a terminal just like in the GUI. Setting the size is not possible in most terminals.

You can start the X-Windows version of gvim with an argument to specify the size and position of the window:

gvim -geometry {width}x{height}+{x_offset}+{y_offset}

{width} and **{height}** are in characters, {x_offset} and {y_offset} are in pixels. Example:

gvim -geometry 80x25+100+300

227. Various

- **gvim -f file.txt**
  The "-f" stands for foreground. Now Vim will block the shell it was started in until you finish editing and exit.

- If you are editing a file and decide you want to use the GUI after all, you can start it with:
  :gui
- If for some reason you don't want to use the normal gvimrc file, you can specify another one with the "-U" argument:
  gvim -U thisrc ...

The undo tree

Vim provides multi-level undo. If you undo a few changes and then make a new change you create a branch in the undo tree. This text is about moving through the branches.

228. Undo up to a file write

Sometimes you make several changes, and then discover you want to go back to when you have last written the file. You can do that with this command:

```
:earlier 1f
```

The "f" stands for "file" here.

You can repeat this command to go further back in the past. Or use a count different from 1 to go back faster.

If you go back too far, go forward again with:

```
:later 1f
```

Note that these commands really work in time sequence. This matters if you made changes after undoing some changes. It's explained in the next section.

Also note that we are talking about text writes here. For writing the undo information in a file see **|undo-persistence|.**

229. Numbering changes

In section |02.5| we only discussed one line of undo/redo. But it is also possible to branch off. This happens when you undo a few changes and then make a new change. The new changes become a branch in the undo tree.

Let's start with the text "one". The first change to make is to append " too". And then move to the first 'o' and change it into 'w'. We then have two changes, numbered 1 and 2, and three states of the text:

```
          one
           |
        change 1
           |
        one too
           |
        change 2
           |
        one two
```

If we now undo one change, back to "one too", and change "one" to "me" we create a branch in the undo tree:

```
          one
           |
        change 1
           |
        one too
        /     \
   change 2  change 3
      |         |
   one two    me too
```

You can now use the |u| command to undo. If you do this twice you get to "one". Use |CTRL-R| to redo, and you will go to "one too". One more **|CTRL-R|** takes you to "me too". **Thus undo and redo go up and down in the tree, using the branch that was last used.**

230. Jumping around the tree

So how do you get to "one two" now? You can use this command:

```
:undo 2
```

The text is now "one two", you are below change 2. You can use the **|:undo|** command to jump to below any change in the tree.

Now make another change: change "one" to "not":

```
          one
           |
        change 1
           |
        one too
        /     \
   change 2  change 3
      |         |
   one two    me too
      |
   change 4
      |
   not two
```

Now you change your mind and want to go back to "me too". Use the **|g-|** command. This moves back in time. Thus it doesn't walk the tree upwards or downwards, but goes to the change made before.

You can repeat **|g-|** and you will see the text change:

```
  me too
  one two
  one too
  one
```

Use **|g+|** to move forward in time:

```
  one
  one too
  one two
  me too
  not two
```

Using |:undo| is useful if you know what change you want to jump to. **|g-| and |g+| are useful if you don't know exactly what the change number is.**

You can type **a count before |g-| and |g+|** to repeat them.

231. Time travelling

When you have been working on text for a while the tree grows to become big. Then you may want to go to the text of **some minutes ago.**

To see what branches there are in the undo tree use this command:

:undolist

```
number changes  time
    3       2  16 seconds ago
    4       3  5 seconds ago
```

Here you can see the number of the leaves in each branch and when the change was made. Assuming we are below change 4, at "not two", you can go back ten seconds with this command:

:earlier 10s

Depending on how much time you took for the changes you end up at a certain position in the tree. The |:earlier| command argument can be **"m" for minutes, "h" for hours and "d" for days.** To go all the way back use a big number:

:earlier 100d

To travel forward in time again use the **|:later|** command:

:later 1m

The arguments are "s", "m" and "h", just like with |:earlier|.

If you want even more details, or want to manipulate the information, you can use the **undotree()** function. To see what it returns:

:echo undotree()

Make new commands

Vim is an extensible editor. You can take a sequence of commands you use often and turn it into a new command. Or redefine an existing command. Autocommands make it possible to execute commands automatically.

232. Key mapping

:map <F2> GoDate: <Esc>:read !date<CR>kJ

The ":read !date" command reads the output from the "date" command and appends it below the current line. The <CR> is required to execute the ":read" command.

- **MAPPING AND MODES**

Here is an overview of map commands and in which mode they work:

:map	Normal, Visual and Operator-pending
:vmap	Visual
:nmap	Normal
:omap	Operator-pending
:map!	Insert and Command-line
:imap	Insert
:cmap	Command-line

Operator-pending mode is when you typed an operator character, such as "d" or "y", and you are expected to type the motion command or a text object. Thus when you type "dw", the "w" is entered in operator-pending mode.

Suppose that you want to define <F7> so that the command d<F7> deletes a C program block (text enclosed in curly braces, {}). Similarly y<F7> would yank the program block into the unnamed register. Therefore, what you need to do is to define <F7> to select the current program block. You can do this with the following command:

```
:omap <F7> a{
```

- **LISTING MAPPINGS**

To see the currently defined mappings, use **":map"** without arguments.

- **REMAPPING**

The result of a mapping is inspected for other mappings in it. For example, the mappings for <F2> above could be shortened to:

```
:map <F2> G<F3>
:imap <F2> <Esc><F3>
:map <F3>  oDate: <Esc>:read !date<CR>kJ
```

For Normal mode <F2> is mapped to go to the last line, and then behave like <F3> was pressed. In Insert mode <F2> stops Insert mode with <Esc> and then also uses <F3>. Then <F3> is mapped to do the actual work.

Suppose you hardly ever use Ex mode, and want to use the "Q" command to format text (this was so in old versions of Vim). This mapping will do it:

```
:map Q gq
```

But, in rare cases you need to use Ex mode anyway. Let's map "gQ" to Q, so that you can still go to Ex mode:

```
:map gQ Q
```

What happens now is that when you type "gQ" it is mapped to "Q". So far so good. But then "Q" is mapped to "gq", thus typing "gQ" results in "gq", and you don't get to Ex mode at all.

To avoid keys to be mapped again, use the ":noremap" command:

```
:noremap gQ Q
```

Now Vim knows that the "Q" is not to be inspected for mappings that apply to it. There is a similar command for every mode:

:noremap	Normal, Visual and Operator-pending
:vnoremap	Visual
:nnoremap	Normal
:onoremap	Operator-pending
:noremap!	Insert and Command-line
:inoremap	Insert
:cnoremap	Command-line

- **RECURSIVE MAPPING**

  When a mapping triggers itself, it will run forever. This can be used to repeat an action an unlimited number of times.

  For example, you have a list of files that contain a version number in the first line. You edit these files with "vim *.txt". You are now editing the first file. Define this mapping:

  ```
  :map ,, :s/5.1/5.2/<CR>:wnext<CR>,,
  ```

  When a mapping runs into an error halfway, the rest of the mapping is **discarded**. **CTRL-C** interrupts the mapping (CTRL-Break on MS-Windows).

- **DELETE A MAPPING**

  To remove a mapping use the **":unmap"** command. Again, the mode the unmapping applies to depends on the command used:

:unmap	Normal, Visual and Operator-pending
:vunmap	Visual

:nunmap Normal
:ounmap Operator-pending
:unmap! Insert and Command-line
:iunmap Insert
:cunmap Command-line

There is a trick to define a mapping that works in Normal and Operator-pending mode, but not in Visual mode. First define it for all three modes, then delete it for Visual mode:

```
:map <C-A> /---><CR>
:vunmap <C-A>
```

Notice that the five characters **"<C-A>"** stand for the single key **CTRL-A.**

To remove all mappings use the **|:mapclear|** command. You can guess the variations for different modes by now. Be careful with this command, it can't be undone.

- **SPECIAL CHARACTERS**

The ":map" command can be followed by another command. A **|** character separates the two commands. This also means that a | character can't be used inside a map command. To include one, use <Bar> (five characters). Example:

```
:map <F8> :write <Bar> !checkin %<CR>
```

The same problem applies to the ":unmap" command, with the addition that you have to watch out for trailing white space. These two commands are different:

```
:unmap a | unmap b
:unmap a| unmap b
```

The first command tries to unmap **"a ", with a trailing space.**

When using a space inside a mapping, use <Space> (seven characters):

```
:map <Space> W
```

This makes the spacebar move a blank-separated word forward.

It is not possible to put a comment directly after a mapping, because the " character is considered to be part of the mapping. You can use |", this starts a new, empty command with a comment. Example:

```
:map <Space> W|     " Use spacebar to move forward a word
```

- **MAPPINGS AND ABBREVIATIONS**

:imap aa foo

:imap aaa bar

- **ADDITIONALLY**

  The <script> keyword can be used to make a mapping local to a script. See **|:map-<script>|.**

  The <buffer> keyword can be used to make a mapping local to a specific buffer. See **|:map-<buffer>|**

  The <unique> keyword can be used to make defining a new mapping fail when it already exists. Otherwise a new mapping simply overwrites the old one. See **|:map-<unique>|**.

  To make a key do nothing, map it to **<Nop>** (five characters). This will make the **<F7>** key do nothing at all:

  :map <F7> <Nop>| map! <F7> <Nop>

  There must be **no space after <Nop>.**

233. Defining command-line commands

You execute these commands just like any other Command-line mode command.

To define a command, use the ":command" command, as follows:

:command DeleteFirst 1delete

Now when you execute the command ":DeleteFirst" Vim executes ":1delete", which deletes the first line.

- **NUMBER OF ARGUMENTS**

  User-defined commands can take a series of arguments. The number of arguments must be specified by the -nargs option. For instance, the example :DeleteFirst command takes no arguments, so you could have defined it as follows:

  :command -nargs=0 DeleteFirst 1delete

  However, because zero arguments is the default, you do not need to add "-nargs=0". The other values of -nargs are as follows:

-nargs=0	No arguments
-nargs=1	One argument
-nargs=*	Any number of arguments
-nargs=?	Zero or one argument

-nargs=+	One or more arguments

- **USING THE ARGUMENTS**

  Inside the command definition, the arguments are represented by the **<args>** keyword. For example:

  ```
  :command -nargs=+ Say :echo "<args>"
  ```

  Now when you type

  ```
  :Say Hello World
  ```

  Vim echoes "Hello World". However, if you add a double quote, it won't work.
  For example:

  ```
  :Say he said "hello"
  ```

  To get special characters turned into a string, properly escaped to use as an expression, use **"<q-args>"**:

  ```
  :command -nargs=+ Say :echo <q-args>
  ```

  Now the above ":Say" command will result in this to be executed:

  ```
  :echo "he said \"hello\""
  ```

  The **<f-args>** keyword contains the same information as the <args> keyword, except in a format suitable for use as function call arguments. For example:

  ```
  :command -nargs=* DoIt :call AFunction(<f-args>)
  :DoIt a b c
  ```

  Executes the following command:

  ```
  :call AFunction("a", "b", "c")
  ```

- **LINE RANGE**

  Some commands take a range as their argument. To tell Vim that you are defining such a command, you need to specify a -range option. The values for this option are as follows:

-range	Range is allowed; default is the current line.
-range=%	Range is allowed; default is the whole file.
-range={count}	Range is allowed; the last number in it is used as a single number whose default is {count}.

When a range is specified, the keywords **<line1>** and **<line2>** get the values of the first and last line in the range. For example, the following command defines the SaveIt command, which writes out the specified range to the file "save_file":

```
:command -range=% SaveIt :<line1>,<line2>write! save_file
```

- **OTHER OPTIONS**

  Some of the other options and keywords are as follows:

-count={number}	The command can take a count whose default is {number}. The resulting count can be used through the <count> keyword.
-bang	You can use a !. If present, using <bang> will result in a !.
-register	You can specify a register. (The default is the unnamed register.) The register specification is available as <reg> (a.k.a. <register>).
-complete={type}	Type of command-line completion used. See |:command-completion| for the list of possible values.
-bar	The command can be followed by | and another command, or " and a comment.
-buffer	The command is only available for the current buffer.

  Finally, you have the **<lt>** keyword. It stands for the character <. Use this to escape the special meaning of the <> items mentioned.

- **REDEFINING AND DELETING**

  To redefine the same command use the ! argument:

```
:command -nargs=+ Say :echo "<args>"
:command! -nargs=+ Say :echo <q-args>
```

  To delete a user command use ":delcommand". It takes a single argument, which is the name of the command. Example:

```
:delcommand SaveIt
```

To delete all the user commands:

```
:comclear
```

Careful, this can't be undone!
More details about all this in the reference manual: **|user-commands|.**

234. Autocommands

An autocommand is a command that is executed automatically in response to some event, such as a file being read or written or a buffer change. Through the use of autocommands you can train Vim to edit compressed files, for example. That is used in the |gzip| plugin.

Autocommands are very powerful. Use them with care and they will help you avoid typing many commands. Use them carelessly and they will cause a lot of trouble.

Suppose you want to replace a datestamp on the end of a file every time it is written. First you define a function:

```
:function DateInsert()
:  $delete
:  read !date
:endfunction
```

You want this function to be called each time, just before a file is written. This will make that happen:

```
:autocmd FileWritePre *  call DateInsert()
```

"FileWritePre" is the event for which this autocommand is triggered: Just before (pre) writing a file. The **"*"** is a pattern to match with the file name. In this case it matches all files.

With this command enabled, when you do a ":write", Vim checks for any matching FileWritePre autocommands and executes them, and then it performs the ":write".

The general form of the :autocmd command is as follows:

```
:autocmd [group] {events} {file_pattern} [nested] {command}
```

The **[group]** name is optional. It is used in managing and calling the commands (more on this later). The **{events}** parameter is a list of events (comma separated) that trigger the command.

{file_pattern} is a filename, usually with wildcards. For example, using "*.txt" makes the autocommand be used for all files whose name end in ".txt".

The optional [nested] flag allows for nesting of autocommands (see below), and finally, {command} is the command to be executed.

- **EVENTS**
  One of the most useful events is **BufReadPost**. It is triggered after a new file is being edited. It is commonly used to set option values. For example, you know that "*.gsm" files are GNU assembly language. To get the syntax file right, define this autocommand:

```
:autocmd BufReadPost *.gsm  set filetype=asm
```

  If Vim is able to detect the type of file, it will set the **'filetype'** option for you. This triggers the Filetype event. Use this to do something when a certain type of file is edited. For example, to load a list of abbreviations for text files:

```
:autocmd Filetype text  source ~/.vim/abbrevs.vim
```

  When starting to edit a new file, you could make Vim insert a skeleton:

```
:autocmd BufNewFile *.[ch]  0read ~/skeletons/skel.c
```

  See **|autocmd-events|** for a complete list of events.

- **PATTERNS**
  The {file_pattern} argument can actually be a comma-separated list of file patterns. For example: "*.c,*.h" matches files ending in ".c" and ".h".
  The usual file wildcards can be used. Here is a summary of the most often used ones:

*****	Match any character any number of times
?	Match any character once
[abc]	Match the character a, b or c
.	Matches a dot
a{b,c}	Matches "ab" and "ac"

  When the pattern includes a slash (/) Vim will compare directory names. Without the slash only the last part of a file name is used. For example, "*.txt" matches "/home/biep/readme.txt". The pattern "/home/biep/*" would also match it. But "home/foo/*.txt" wouldn't.

When including a slash, Vim matches the pattern against both the full path of the file ("/home/biep/readme.txt") and the relative path (e.g., "biep/readme.txt").

> **Note:**
> When working on a system that uses a backslash as file separator, such as MS-Windows, you still use forward slashes in autocommands. This makes it easier to write the pattern, since a backslash has a special meaning. It also makes the autocommands portable.

- **DELETING**
  To delete an autocommand, use the same command as what it was defined with, but leave out the {command} at the end and use a !. Example:
  ```
  :autocmd! FileWritePre *
  ```
  This will delete all autocommands for the "FileWritePre" event that use the "*" pattern.

- **LISTING**
  To list all the currently defined autocommands, use this:
  ```
  :autocmd
  ```
  The list can be very long, especially when filetype detection is used. To list only part of the commands, specify the group, event and/or pattern. For example, to list all BufNewFile autocommands:
  ```
  :autocmd BufNewFile
  ```
  To list all autocommands for the pattern "*.c":
  ```
  :autocmd * *.c
  ```
  Using "*" for the event will list all the events. To list all autocommands for the cprograms group:
  ```
  :autocmd cprograms
  ```

- **GROUPS**

The **{group}** item, used when defining an autocommand, groups related autocommands together. This can be used to delete all the autocommands in a certain group, for example.
When defining several autocommands for a certain group, use the ":augroup" command. For example, let's define autocommands for C programs:

```
:augroup cprograms
:  autocmd BufReadPost *.c,*.h :set sw=4 sts=4
:  autocmd BufReadPost *.cpp   :set sw=3 sts=3
:augroup END
```

This will do the same as:

```
:autocmd cprograms BufReadPost *.c,*.h :set sw=4 sts=4
:autocmd cprograms BufReadPost *.cpp   :set sw=3 sts=3
```

To delete all autocommands in the "cprograms" group:

```
:autocmd! cprograms
```

- **NESTING**
  Generally, commands executed as the result of an autocommand event will not trigger any new events. If you read a file in response to a FileChangedShell event, it will not trigger the autocommands that would set the syntax, for example. To make the events triggered, add the **"nested"** argument:

  ```
  :autocmd FileChangedShell * nested  edit
  ```

- **EXECUTING AUTOCOMMANDS**
  It is possible to trigger an autocommand by pretending an event has occurred. This is useful to have one autocommand trigger another one. Example:

  ```
  :autocmd BufReadPost *.new  execute "doautocmd BufReadPost " . expand("<afile>:r")
  ```

  This defines an autocommand that is triggered when a new file has been edited. The file name must end in ".new". The "**:execute**" command uses expression evaluation to form a new command and execute it. When editing the file "tryout.c.new" the executed command will be:

  ```
  :doautocmd BufReadPost tryout.c
  ```

The expand() function takes the **"<afile>"** argument, which stands for the file name the autocommand was executed for, and takes the root of the file name with ":r". ":doautocmd" executes on the current buffer. The ":doautoall" command works like "doautocmd" except it executes on all the buffers.

- **USING NORMAL MODE COMMANDS**

The commands executed by an autocommand are **Command-line** commands. If you want to use a Normal mode command, the **":normal"** command can be used. Example:

```
:autocmd BufReadPost *.log normal G
```

This will make the cursor jump to the last line of *.log files when you start to edit it.
Using the ":normal" command is a bit tricky. First of all, make sure its argument is a complete command, including all the arguments. When you use "I" to go to Insert mode, there must also be a **<Esc>** to leave Insert mode again. If you use a "/" to start a search pattern, there must be a **<CR>** to execute it.
The ":normal" command uses all the text after it as commands. Thus there can be no | and another command following. To work around this, put the ":normal" command inside an ":execute" command. This also makes it possible to pass unprintable characters in a convenient way. Example:

```
:autocmd BufReadPost *.chg execute "normal ONew entry:\<Esc>" |
    \ 1read !date
```

This also shows the use of a **backslash** to break a long command into **more lines**. This can be used in Vim scripts (not at the command line). The output as follow:

```
New entry:
Wed Sep 19 12:48:07 CST 2012
```

When you want the autocommand do something complicated, which involves jumping around in the file and then returning to the original position, you may want to restore the view on the file. See |restore-position| for an example.

- **IGNORING EVENTS**

At times, you will not want to trigger an autocommand. The **'eventignore'** option contains a list of events that will be totally ignored. For example, the following causes events for entering and leaving a window to be ignored:

```
:set eventignore=WinEnter,WinLeave
```

To ignore all events, use the following command:

```
:set eventignore=all
```

To set it back to the normal behavior, make 'eventignore' empty:

```
:set eventignore=
```

Write a Vim script

The Vim script language is used for the startup vimrc file, syntax files, and many other things. This chapter explains the items that can be used in a Vim script. There are a lot of them, thus this is a long chapter.

235. Introduction

Your first experience with Vim scripts is the vimrc file. Vim reads it when it starts up and executes the commands. You can set options to values you prefer. And you can use any colon command in it (commands that start with a ":"; these are sometimes referred to as Ex commands or command-line commands).

Syntax files are also Vim scripts. As are files that set options for a specific file type. A complicated **macro** can be defined by a separate Vim script file. You can think of other uses yourself.

A simple example:

```
:let i = 1
:while i < 5
:  echo "count is" i
:  let i += 1
:endwhile
```

It can be written much more compact:

```
:for i in range(1, 4)
```

```
:  echo "count is" i
:endfor
```

- **THREE KINDS OF NUMBERS**

  Numbers can be decimal, hexadecimal or octal. A hexadecimal number starts with "0x" or "0X". For example "0x1f" is decimal 31. An octal number starts with a zero. "017" is decimal 15.

  The ":echo" command always prints decimal numbers. Example:

  ```
  :echo 0x7f 036
  127 30
  ```

  A number is made negative with a minus sign.

  ```
  :echo 0x7f -036
  97
  ```

236. Variables

These variables are global. To see a list of currently defined variables use this command:

```
:let
```

You can use global variables everywhere. This also means that when the variable "count" is used in one script file, it might also be used in another file. This leads to confusion at least, and real problems at worst. To avoid this, you can use a variable local to a script file by prepending **"s:"**. For example, one script contains this code:

```
:let s:count = 1
:while s:count < 5
:  source other.vim
:  let s:count += 1
:endwhile
```

There are more kinds of variables, see **|internal-variables|.** The most often used ones are:

b:name	variable local to a buffer
w:name	variable local to a window
g:name	global variable (also in a function)
v:name	variable predefined by Vim

- **DELETING VARIABLES**

  ```
  :unlet s:count
  ```

  This deletes the script-local variable "s:count" to free up the memory it uses. If you are not sure if the variable exists, and don't want an error message when it doesn't, append !:

:unlet! s:count

When a script finishes, the local variables used there will not be automatically freed. The next time the script executes, it can still use the old value.

- **STRING VARIABLES AND CONSTANTS**

  Numbers and strings are the basic types of variables that Vim supports. The type is dynamic, it is set each time when assigning a value to the variable with ":let". More about types in |41.8|.

  :let name = "peter"
  :let name = "\"peter\""
  :let name = '"peter"'

  In double-quote strings it is possible to use special characters. Here are a few useful ones:

\t	<Tab>
\n	<NL>, line break
\r	<CR>, <Enter>
\e	<Esc>
\b	<BS>, backspace
\"	"
\\	\\, backslash
\\<Esc>	<Esc>
\\<C-W>	CTRL-W

  The last two are just examples. The **"\<name>"** form can be used to include the special key "name".

  See **|expr-quote|** for the full list of special items in a string.

237. Expressions

You can read the definition here: |expression-syntax|.

The numbers, strings and variables mentioned above are expressions by themselves. Thus everywhere an expression is expected, you can use a number, string or variable. Other basic items in an expression are:

$NAME	environment variable
&name	option
@r	register

The **&name** form can be used to save an option value, set it to a new value, do something and restore the old value. Example:

```
:let save_ic = &ic
:set noic
:/The Start/,$delete
:let &ic = save_ic
```

This makes sure the "The Start" pattern is used with the 'ignorecase' option off. Still, it keeps the value that the user had set. (Another way to do this would be to add "\C" to the pattern, see |/\C|.)

- **MATHEMATICS**

a + b	add
a - b	subtract
a * b	multiply
a / b	divide
a % b	modulo

Strings can be concatenated with ".". Example:

```
:echo "foo" . "bar"
foobar
```

If "a" evaluates to true "b" is used, otherwise "c" is used. Example:

```
:let i = 4
:echo i > 5 ? "i is big" : "i is small"
i is small
```

- **Conditionals**

```
:if {condition}
          {statements}
  :elseif {condition}
          {statements}
 :else {condition}
          {statements}
:endif
```

- **LOGIC OPERATIONS**

a == b	equal to
a != b	not equal to
a > b	greater than

a >= b	greater than or equal to
a < b	less than
a <= b	less than or equal to

The logic operators work both for numbers and strings. When comparing two strings, the mathematical difference is used. This compares byte values, which may not be right for some languages.

For strings there are two more items:

a =~ b	matches with
a !~ b	does not match with

The 'ignorecase' option is used when comparing strings. When you don't want that, append **"#"** to match case and **"?"** to ignore case. Thus "==?" compares two strings to be equal while ignoring case. And "!~#" checks if a pattern doesn't match, also checking the case of letters. For the full table see **|expr-==|.**

- **MORE LOOPING**

The **":while"** command was already mentioned. Two more statements can be used in between the ":while" and the ":endwhile":

:continue	Jump back to the start of the while loop; the loop continues.
:break	Jump forward to the ":endwhile"; the loop is discontinued.

The ":sleep" command makes Vim take a nap. The "50m" specifies fifty milliseconds. Another example is ":sleep 4", which sleeps for four seconds.

Even more looping can be done with the **":for"** command, see below in |41.8|.

238. Executing an expression

The ":execute" command allows executing the result of an expression.

:execute "tag " . tag_name

The **"."** is used to concatenate the string "tag " with the value of variable "tag_name". Suppose "tag_name" has the value "get_cmd", then the command that will be executed is:

:tag get_cmd

The **":execute"** command can only execute colon commands. The **":normal"** command executes Normal mode commands. However, its argument is not an expression but the literal command characters. Example:

```
:normal gg=G
```

This jumps to the first line and formats all lines with the **"="** operator.

To make ":normal" work with an expression, combine ":execute" with it. Example:

```
:execute "normal " . normal_commands
:execute "normal Inew text \<Esc>"
```

Notice the use of the special key "\<Esc>". This avoids having to enter a real <Esc> character in your script.

If you don't want to execute a string but evaluate it to get its expression value, you can use the eval() function:

```
:let optname = "path"
:let optval = eval('&' . optname)
```

A "&" character is prepended to "path", thus the argument to eval() is "&path". The result will then be the value of the 'path' option.

The same thing can be done with:

```
:exe 'let optval = &' . optname
```

239. Using functions

A function is called with the **":call"** command. The parameters are passed in between parentheses separated by commas. Example:

```
:call search("Date: ", "W")
```

You can find the whole list here: **|functions|.**

- **:help function-list**

240. Defining a function

Vim enables you to define your own functions. The basic function declaration begins as follows:

```
:function {name}({var1}, {var2}, ...)
:  {body}
:endfunction
```

Note:

Function names must begin with **a capital letter**.

The complete function definition is as follows:

```
:function Min(num1, num2)
:  if a:num1 < a:num2
:    let smaller = a:num1
:  else
:    let smaller = a:num2
:  endif
:  return smaller
:endfunction
```

A user defined function is called in exactly the same way as a built-in function. Only the name is different. The Min function can be used like this:

```
:echo Min(5, 8)
```

When a function reaches ":endfunction" or ":return" is used without an argument, the function returns zero.

To redefine a function that already exists, use the ! for the ":function" command:

```
:function!  Min(num1, num2, num3)
```

- **USING A RANGE**

  The ":call" command can be given a line range. This can have one of two meanings. When a function has been defined with the "range" keyword, it will take care of the line range itself.

  The function will be passed the variables "a:firstline" and "a:lastline". These will have the line numbers from the range the function was called with. Example:

```
:function Count_words() range
:  let lnum = a:firstline
:  let n = 0
:  while lnum <= a:lastline
:    let n = n + len(split(getline(lnum)))
:    let lnum = lnum + 1
:  endwhile
:  echo "found " . n . " words"
:endfunction
```

You can call this function with:

```
:10,30call Count_words()
```

It will be executed **once** and echo the number of words.

The other way to use a line range is by defining a function without the "range" keyword. The function will be called once for every line in the range, with the cursor in that line. Example:

```
:function  Number()
:  echo "line " . line(".") . " contains: " . getline(".")
:endfunction
```

If you call this function with:

```
:10,15call Number()
```

The function will be called **six times.**

- **VARIABLE NUMBER OF ARGUMENTS**

  Vim enables you to define functions that have a variable number of arguments. The following command, for instance, defines a function that must have 1 argument (start) and can have up to 20 additional arguments:

  ```
  :function Show(start, ...)
  ```

  The variable **"a:1"** contains the first optional argument, "a:2" the second, and so on. The variable **"a:0"** contains the number of extra arguments.

  You can also use the **a:000** variable, it is a List of all the "..." arguments. See **|a:000|**.

- **LISTING FUNCTIONS**

  The **":function"** command lists the names and arguments of all user-defined functions: To see what a function does, use its name as an argument for ":function".

- **DEBUGGING**

  The line number is useful for when you get an error message or when debugging. See **|debug-scripts|** about debugging mode.

  You can also set the **'verbose'** option to 12 or higher to see all function calls. Set it to 15 or higher to see every executed line.

- **DELETING A FUNCTION**

  To delete the Show() function:

  ```
  :delfunction Show
  ```

- **FUNCTION REFERENCES**

Sometimes it can be useful to have a variable point to one function or another. You can do it with the f**unction()** function. **It turns the name of a function into a reference:**

```
:let result = 0         " or 1
:function! Right()
:  return 'Right!'
:endfunc
:function! Wrong()
:  return 'Wrong!'
:endfunc
:
:if result == 1
:  let Afunc = function('Right')
:else
:  let Afunc = function('Wrong')
:endif
:echo call(Afunc, [])
Wrong!
```

Note that the name of a variable that holds a function reference must start with a capital. Otherwise it could be confused with the name of a builtin function.

The way to invoke a function that a variable refers to is with the **call()** function. Its first argument is **the function reference**, the second argument is a List with arguments.

Function references are most useful in combination with a Dictionary, as is explained in the next section.

241. Lists and Dictionaries

A **List** is an ordered sequence of things. The things can be any kind of value, thus you can make a List of numbers, a List of Lists and even a List of mixed items. To create a List with three strings:

```
:let alist = ['aap', 'mies', 'noot']
```

The List items are enclosed in square brackets and separated by commas. To create an empty List:

```
:let alist = []
```

You can add items to a List with the **add()** function:

```
:let alist = []
:call add(alist, 'foo')
:call add(alist, 'bar')
:echo alist
['foo', 'bar']
```

List concatenation is done with +:

```
:echo alist + ['foo', 'bar']
['foo', 'bar', 'foo', 'bar']
```

Or, if you want to extend a List directly:

```
:let alist = ['one']
:call extend(alist, ['two', 'three'])
:echo alist
['one', 'two', 'three']
```

Notice that using add() will have a different effect:

```
:let alist = ['one']
:call add(alist, ['two', 'three'])
:echo alist
['one', ['two', 'three']]
```

The second argument of add() is added as a single item.

- **FOR LOOP**

One of the nice things you can do with a List is **iterate** over it:

```
:let alist = ['one', 'two', 'three']
:for n in alist
:  echo n
:endfor
one
two
three
```

To loop a certain number of times you need a List of a specific length. The **range()** function creates one for you:

```
:for a in range(3)
:  echo a
:endfor
0
1
2
```

You can also specify the maximum value, the stride and even go backwards:

```
:for a in range(8, 4, -2)
:  echo a
:endfor
8
6
4
```

A more useful example, looping over lines in the buffer:

```
:for line in getline(1, 20)
:  if line =~ "Date: "
:    echo matchstr(line, 'Date: \zs.*')
:  endif
:endfor
```

This looks into lines 1 to 20 (inclusive) and echoes any date found in there.

- **DICTIONARIES**

A **Dictionary** stores key-value pairs. You can quickly lookup a **value** if you know the **key**.

A Dictionary is created with curly braces:

```
:let uk2nl = {'one': 'een', 'two': 'twee', 'three': 'drie'}
```

Now you can lookup words by putting the key in square brackets:

```
:echo uk2nl['two']
twee
```

The generic form for defining a Dictionary is:

{<key> : <value>, ...}

The possibilities with Dictionaries are numerous. There are various functions for them as well. For example, you can obtain a list of the keys and loop over them:

```
:for key in keys(uk2nl)
:  echo key
```

```
:endfor
three
one
two
```

You will notice the keys are not ordered. You can sort the list to get a specific order:

```
:for key in sort(keys(uk2nl))
:  echo key
:endfor
one
three
two
```

But you can never get back the order in which items are defined. For that you need to use a List, it stores items in an ordered sequence.

- **DICTIONARY FUNCTIONS**

The items in a Dictionary can normally be obtained with an index in square brackets:

```
:echo uk2nl['one']
een
```

A method that does the same, but without so many punctuation characters:

```
:echo uk2nl.one
een
```

This only works for a key that is made of ASCII letters, digits and the underscore. You can also assign a new value this way:

```
:let uk2nl.four = 'vier'
:echo uk2nl
{'three': 'drie', 'four': 'vier', 'one': 'een', 'two': 'twee'}
```

And now for something special: you can directly define a function and store a reference to it in the dictionary:

```
:function uk2nl.translate(line) dict
:  return join(map(split(a:line), 'get(self, v:val, "???")'))
:endfunction
```

Let's first try it out:

```
:echo uk2nl.translate('three two five one')
drie twee ??? een
```

The first special thing you notice is the **"dict"** at the end of the ":function" line. **This marks the function as being used from a Dictionary.** The "self" local variable will then refer to that Dictionary.

Now let's break up the complicated return command:

```
split(a:line)
```

The split() function takes a string, chops it into whitespace separated words and returns a list with these words. Thus in the example it returns:

```
:echo split('three two five one')
['three', 'two', 'five', 'one']
```

This list is the first argument to the **map()** function. This will go through the list, evaluating its second argument with "v:val" set to the value of each item. This is a shortcut to using a for loop. This command:

```
:let alist = map(split(a:line), 'get(self, v:val, "???")')
```

Is equivalent to:

```
:let alist = split(a:line)
:for idx in range(len(alist))
:  let alist[idx] = get(self, alist[idx], "???")
:endfor
```

The **get()** function checks if a key is present in a Dictionary. If it is, then the value is retrieved. If it isn't, then the default value is returned, in the example it's '???'. This is a convenient way to handle situations where a key may not be present and you don't want an error message.

The join() function does the opposite of split(): it joins together a list of words, putting a space in between.

This combination of split(), map() and join() is a nice way to filter a line of words in a very compact way.

- **OBJECT ORIENTED PROGRAMMING**

You can actually use a Dictionary like an object.
Above we used a Dictionary for translating Dutch to English. We might want to do the same for other languages. Let's first make an object (aka Dictionary) that has the translate function, but no words to translate:

```
:let transdict = {}
:function transdict.translate(line) dict
:  return join(map(split(a:line), 'get(self.words, v:val, "???")'))
:endfunction
```

It's slightly different from the function above, using 'self.words' to lookup word translations. But we don't have a self.words. Thus you could call this an abstract class.

Now we can instantiate a Dutch translation object:

```
:let uk2nl = copy(transdict)
:let uk2nl.words = {'one': 'een', 'two': 'twee', 'three': 'drie'}
:echo uk2nl.translate('three one')
drie een
```

And a German translator:

```
:let uk2de = copy(transdict)
:let uk2de.words = {'one': 'ein', 'two': 'zwei', 'three': 'drei'}
:echo uk2de.translate('three one')
drei ein
```

You see that the copy() function is used to make a copy of the "transdict" Dictionary and then the copy is changed to add the words. The original remains the same, of course.

Now you can go one step further, and use your preferred translator:

```
:if $LANG =~ "de"
:  let trans = uk2de
:else
:  let trans = uk2nl
:endif
```

```
:echo trans.translate('one two three')
een twee drie
```

Here "trans" refers to one of the two objects (Dictionaries). No copy is made. More about List and Dictionary identity can be found at |list-identity| and |dict-identity|.

Now you might use a language that isn't supported. You can overrule the translate() function to do nothing:

```
:let uk2uk = copy(transdict)
:function! uk2uk.translate(line)
:  return a:line
:endfunction
:echo uk2uk.translate('three one wladiwostok')
three one wladiwostok
```

Notice that a ! was used to overwrite the existing function reference. Now use "uk2uk" when no recognized language is found:

```
:if $LANG =~ "de"
:  let trans = uk2de
:elseif $LANG =~ "nl"
:  let trans = uk2nl
:else
:  let trans = uk2uk
:endif
:echo trans.translate('one two three')
one two three
```

For further reading see **|Lists|** and **|Dictionaries|**.

242. Exceptions

Let's start with an example:

```
:try
:   read ~/templates/pascal.tmpl
:catch /E484:/
:   echo "Sorry, the Pascal template file cannot be found."
```

```
:endtry
```

The ":read" command will fail if the file does not exist. Instead of generating an error message, this code catches the error and gives the user a nice message.

For the commands in between ":try" and ":endtry" errors are turned into exceptions. An exception is a string. In the case of an error the string contains the error message. And every error message has a number. In this case, the error we catch contains "E484:". This number is guaranteed to stay the same (the text may change, e.g., it may be translated).

When the ":read" command causes another error, the pattern "E484:" will not match in it. Thus this exception will not be caught and result in the usual error message.

You might be tempted to do this:

```
:try
:   read ~/templates/pascal.tmpl
:catch
:   echo "Sorry, the Pascal template file cannot be found."
:endtry
```

This means all errors are caught. But then you will not see errors that are useful, such as "E21: Cannot make changes, 'modifiable' is off".

Another useful mechanism is the **":finally"** command:

```
:let tmp = tempname()
:try
:   exe ".,$write " . tmp
:   exe "!filter " . tmp
:   .,$delete
:   exe "$read " . tmp
:finally
:   call delete(tmp)
:endtry
```

This filters the lines from the cursor until the end of the file through the "filter" command, which takes a file name argument. **No matter if the filtering works,** something goes wrong in between **":try"** and **":finally"** or the user cancels the filtering by pressing CTRL-C,

the "call delete(tmp)" is always executed. This makes sure you don't leave the temporary file behind.

More information about **exception handling** can be found in the reference manual: **|exception-handling|.**

243. Various remarks

Here is a summary of items that apply to Vim scripts. They are also mentioned elsewhere, but form a nice checklist.

- **WHITE SPACE**
  For a ":set" command involving the "=" (equal) sign, such as in:

  ```
  :set cpoptions    =aABceFst
  ```

  the whitespace immediately before the "=" sign is ignored. But there can be no whitespace after the "=" sign!

  To include a whitespace character in the value of an option, it must be escaped by a "\" (backslash) as in the following example:

  ```
  :set tags=my\ nice\ file
  ```

- **COMMENTS**
  The character " (the double quote mark) starts a comment.
  There is a little **"catch"** with comments for some commands. Examples:

  ```
  :abbrev dev development         " shorthand
  :map <F3> o#include             " insert include
  :execute cmd                    " do it
  :!ls *.c                        " list C files
  ```

  The abbreviation 'dev' will be expanded to **'development   " shorthand'**. The mapping of <F3> will actually be the whole line after the 'o# ....' including the '"' insert include'. The "execute" command will give an error. The "!" command will send everything after it to the shell, causing an error for an unmatched '"' character.
  There can be no comment after ":map", ":abbreviate", ":execute" and "!" commands (there are a few more commands with this restriction). For the ":map", ":abbreviate" and ":execute" commands there is a trick:

  ```
  :abbrev dev development|" shorthand
  ```

```
**:map <F3> o#include|" insert include**
**:execute cmd                  |" do it**
```

With the '|' character the command is separated from the next one. And that next command is only a comment. For the last command you need to do two things: |:execute| and use '|':

```
:exe '!ls *.c'                  |" list C files
```

Notice that there is no white space before the '|' in the abbreviation and mapping. **For these commands, any character until the end-of-line or '|' is included.** As a consequence of this behavior, you don't always see that trailing whitespace is included:

```
:map <F4> o#include
```

To spot these problems, you can set the 'list' option when editing vimrc files.

For Unix there is one special way to comment a line, that allows making a Vim script executable:

```
#!/usr/bin/env vim -S
echo "this is a Vim script"
quit
```

The "#" command by itself lists a line with the line number. Adding an exclamation mark changes it into doing nothing, so that you can add the shell command to execute the rest of the file. |:#!| |-S|

- **PITFALLS**

Even bigger problem arises in the following example:

```
:map ,ab o#include
:unmap ,ab
```

Here the unmap command will not work, because it tries to unmap ",ab ". This does not exist as a mapped sequence. An error will be issued, which is very hard to identify, because the ending whitespace character in ":unmap ,ab " is not visible.

And this is the same as what happens when one uses a comment after an 'unmap' command:

```
:unmap ,ab     " comment
```

Here the comment part will be ignored. However, Vim will try to unmap ',ab   ', which does not exist. Rewrite it as:

```
:unmap ,ab|    " comment
```

- **RESTORING THE VIEW**

  Sometimes you want to make a change and go back to where cursor was. Restoring the relative position would also be nice, so that the same line appears at the top of the window.
  This example yanks the current line, puts it above the first line in the file and then restores the view:

  ```
  map ,p ma"aYHmbgg"aP`bzt`a
  ```

  What this does:

  ```
  ma"aYHmbgg"aP`bzt`a
  ma                    set mark a at cursor position
    "aY                 yank current line into register a
       Hmb              go to top line in window and set mark b there
          gg            go to first line in file
            "aP         put the yanked line above it
              `b        go back to top line in display
                zt      position the text in the window as before
                  `a    go back to saved cursor position
  ```

- **PACKAGING**

  To avoid your function names to interfere with functions that you get from others, use this scheme:
  - Prepend a unique string before each function name. I often use an abbreviation. For example, "OW_" is used for the option window functions.
  - Put the definition of your functions together in a file. Set a global variable to indicate that the functions have been loaded. When sourcing the file again, **first unload the functions.**

  Example:

  ```
  " This is the XXX package

  if exists("XXX_loaded")
    delfun XXX_one
    delfun XXX_two
  endif
  ```

```
function XXX_one(a)
        ... body of function ...
endfun

function XXX_two(b)
        ... body of function ...
endfun

let XXX_loaded = 1
```

244. Writing a plugin

You can write a Vim script in such a way that many people can use it. This is called a plugin. Vim users can drop your script in their plugin directory and use its features right away |add-plugin|. There are actually two types of plugins:

global plugins: For all types of files.

filetype plugins: Only for files of a specific type.

- **NAME**
  First of all you must choose a name for your plugin. The features provided by the plugin should be clear from its name.

- **HEADER**
  Therefore, put a header at the top of your plugin:

```
" Vim global plugin for correcting typing mistakes
" Last Change:  2000 Oct 15
" Maintainer:   Bram Moolenaar <Bram@vim.org>
```

- **NOT LOADING**
  It's possible that a user doesn't always want to load this plugin. Or the system administrator has dropped it in the system-wide plugin directory, but a user has his own plugin he wants to use. Then the user must have a chance to disable loading this specific plugin. This will make it possible:

```
if exists("g:loaded_typecorr")
  finish
endif
let g:loaded_typecorr = 1
```

This also avoids that when the script is loaded twice it would cause error messages for redefining functions and cause trouble for autocommands that are added twice.

The name is recommended to start with "loaded_" and then the file name of the plugin, literally. The "g:" is prepended just to avoid mistakes when using the variable in a function (without "g:" it would be a variable local to the function).

Using **"finish" stops Vim from reading the rest of the file**, it's much quicker than using if-endif around the whole file.

- **MAPPING**

  Now let's make the plugin more interesting: **We will add a mapping that adds a correction for the word under the cursor.** We could just pick a key sequence for this mapping, but the user might already use it for something else. To allow the user to define which keys a mapping in a plugin uses, the **<Leader>** item can be used:

```
  map <unique> <Leader>a  <Plug>TypecorrAdd
```

  The **"<Plug>TypecorrAdd"** thing will do the work, more about that further on.

  The user can set the "mapleader" variable to the key sequence that he wants this mapping to start with. Thus if the user has done:

```
let mapleader = "_"
```

  the mapping will define "_a". If the user didn't do this, the default value will be used, which is a backslash. Then a map for "\a" will be defined.

  Note that **<unique>** is used, this will cause an error message if the mapping already happened to exist. **|:map-<unique>|**

  But what if the user wants to define his own key sequence? We can allow that with this mechanism:

```
if !hasmapto('<Plug>TypecorrAdd')
  map <unique> <Leader>a  <Plug>TypecorrAdd
endif
```

This checks if a mapping to "<Plug>TypecorrAdd" already exists, and only defines the mapping from "<Leader>a" if it doesn't. The user then has a chance of putting this in his vimrc file:

```
map ,c  <Plug>TypecorrAdd
```

Then the mapped key sequence will be ",c" instead of "_a" or "\a".

- **PIECES**

  If a script gets longer, you often want to break up the work in pieces. You can use functions or mappings for this. But you don't want these functions and mappings to interfere with the ones from other scripts. For example, you could define a function Add(), but another script could try to define the same function. To avoid this, we define the function local to the script by prepending it with **"s:"**.

  We will define a function that adds a new typing correction:

```
function s:Add(from, correct)
  let to = input("type the correction for " . a:from . ": ")
  exe ":iabbrev " . a:from . " " . to
..
endfunction
```

  Now we can call the function s:Add() from within this script. If another script also defines s:Add(), it will be local to that script and can only be called from the script it was defined in. There can also be a global Add() function (without the "s:"), which is again another function.

  <SID> can be used with mappings. It generates a script ID, which identifies the current script. In our typing correction plugin we use it like this:

```
noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
..
noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
```

  Thus when a user types "\a", this sequence is invoked:

```
\a  ->  <Plug>TypecorrAdd  ->  <SID>Add  ->  :call <SID>Add()
```

If another script would also map <SID>Add, it would get another script ID and thus define another mapping.

Note that instead of s:Add() we use <SID>Add() here. That is because the mapping is typed by the user, thus outside of the script. The <SID> is translated to the script ID, so that Vim knows in which script to look for the Add() function.

This is a bit complicated, but it's required for the plugin to work together with other plugins. The basic rule is that you use <SID>Add() in mappings and s:Add() in other places (the script itself, autocommands, user commands).

We can also add a menu entry to do the same as the mapping:

```
 26     noremenu <script> Plugin.Add\ Correction      <SID>Add
```

The "Plugin" menu is recommended for adding menu items for plugins. In this case only one item is used. When adding more items, creating a submenu is recommended. For example, "Plugin.CVS" could be used for a plugin that offers CVS operations "Plugin.CVS.checkin", "Plugin.CVS.checkout", etc.

Note that in line 28 ":noremap" is used to avoid that any other mappings cause trouble. Someone may have remapped ":call", for example. In line 24 we also use ":noremap", but we do want "<SID>Add" to be remapped. This is why "<script>" is used here. This only allows mappings which are local to the script. |:map-<script>| The same is done in line 26 for ":noremenu". **|:menu-<script>|**

- **<SID> AND <Plug>**
  Both <SID> and <Plug> are used to avoid that mappings of typed keys interfere with mappings that are only to be used from other mappings. Note thedifference between using <SID> and <Plug>:

  <Plug> is visible outside of the script. It is used for mappings which the user might want to map a key sequence to. <Plug> is a special code that a typed key will never produce. To make it very unlikely that other plugins use the same sequence of characters, use this structure: <Plug> scriptname mapname In our example the scriptname is "Typecorr" and the mapname is "Add". This results in "<Plug>TypecorrAdd". Only the first character of scriptname and mapname is uppercase, so that we can see where mapname starts.

<SID>   is the script ID, a unique identifier for a script. Internally Vim translates <SID> to "<SNR>123_", where "123" can be any number. Thus a function "<SID>Add()" will have a name "<SNR>11_Add()" in one script, and "<SNR>22_Add()" in another. You can see this if you use the ":function" command to get a list of functions. The translation of <SID> in mappings is exactly the same, that's how you can call a script-local function from a mapping.

- **USER COMMAND**

  Now let's add a user command to add a correction:

```
 38    if !exists(":Correct")
 39      command -nargs=1  Correct  :call s:Add(<q-args>, 0)
 40    endif
```

  The user command is defined only if no command with the same name already exists. Otherwise we would get an error here. Overriding the existing user command with **":command!"** is not a good idea, this would probably make the user wonder why the command he defined himself doesn't work. **|:command|**

- **SCRIPT VARIABLES**

  When a variable starts with "s:" it is a script variable. It can only be used inside a script. Outside the script it's not visible. This avoids trouble with using the same variable name in different scripts. The variables will be kept as long as Vim is running. And the same variables are used when sourcing the same script again. |s:var|

  The fun is that these variables can also be used in functions, autocommands and user commands that are defined in the script. In our example we can add a few lines to count **the number of corrections:**

```
 19    let s:count = 4
 ..
 30    function s:Add(from, correct)
 ..
 34      let s:count = s:count + 1
 35      echo s:count . " corrections now"
 36    endfunction
```

  First s:count is initialized to 4 in the script itself. When later the s:Add() function is called, it increments s:count. It doesn't matter from where the function was called, since it has been defined in the script, it will use the local variables from this script.

- **THE RESULT**

  Here is the resulting complete example:

```
" Vim global plugin for correcting typing mistakes
" Last Change:	2000 Oct 15
" Maintainer:	Bram Moolenaar <Bram@vim.org>
" License:	This file is placed in the public domain.

if exists("g:loaded_typecorr")
  finish
endif
let g:loaded_typecorr = 1

let s:save_cpo = &cpo
set cpo&vim

iabbrev teh the
iabbrev otehr other
iabbrev wnat want
iabbrev synchronisation
	\ synchronization
let s:count = 4

if !hasmapto('<Plug>TypecorrAdd')
  map <unique> <Leader>a  <Plug>TypecorrAdd
endif
noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add

noremenu <script> Plugin.Add\ Correction      <SID>Add

noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>

function s:Add(from, correct)
  let to = input("type the correction for " . a:from . ": ")
```

```
32       exe ":iabbrev " . a:from . " " . to
33       if a:correct | exe "normal viws\<C-R>\" \b\e" | endif
34       let s:count = s:count + 1
35       echo s:count . " corrections now"
36     endfunction
37
38     if !exists(":Correct")
39       command -nargs=1  Correct  :call s:Add(<q-args>, 0)
40     endif
41
42     let &cpo = s:save_cpo
```

Line 33 wasn't explained yet. It applies the new correction to the word under the cursor. The |:normal| command is used to use the new abbreviation. Note that mappings and abbreviations are expanded here, even though the function was called from a mapping defined with ":noremap".

Using "unix" for the 'fileformat' option is recommended. The Vim scripts will then work everywhere. Scripts with 'fileformat' set to "dos" do not work on Unix. Also see |:source_crnl|. To be sure it is set right, do this before writing the file:

```
:set fileformat=unix
```

- **DOCUMENTATION** *__write-local-help__*
  It's a good idea to also write some documentation for your plugin. Especially when its behavior can be changed by the user. See |add-local-help| for how they are installed.

- **FILETYPE DETECTION** ***plugin-filetype***
  If your filetype is not already detected by Vim, you should create a filetype detection snippet in a separate file. It is usually in the form of an autocommand that sets the filetype when the file name matches a pattern. Example:

```
au BufNewFile,BufRead *.foo                set filetype=foofoo
```

Write this single-line file as "ftdetect/foofoo.vim" in the first directory that appears in 'runtimepath'. For Unix that would be "~/.vim/ftdetect/foofoo.vim". The convention is to use the name of the filetype for the script name.

You can make more complicated checks if you like, for example to inspect the contents of the file to recognize the language. Also see **|new-filetype|.**

- **SUMMARY** ***plugin-special***

  Summary of special things to use in a plugin:

s:name	Variables local to the script.
<SID>	Script-ID, used for mappings and functions local to the script.
hasmapto()	Function to test if the user already defined a mapping for functionality the script offers.
<Leader>	Value of "mapleader", which the user defines as the keys that plugin mappings start with.
:map <unique>	Give a warning if a mapping already exists.
:noremap <script>	Use only mappings local to the script, not global mappings.
exists(":Cmd")	Check if a user command already exists.

245. Writing a filetype plugin *write-filetype-plugin* *ftplugin*

246. Writing a compiler plugin *write-compiler-plugin*

247. Writing a plugin that loads quickly *write-plugin-quickload*

248. Writing library scripts *write-library-script*

249. Writing library scripts *write-library-script*

Vim users will look for scripts on the Vim website: http://www.vim.org. If you made something that is useful for others, share it! Vim scripts can be used on any system. There might not be a tar or gzip command. If you want to pack files together and/or compress them the "zip" utility is recommended.

For utmost portability use Vim itself to pack scripts together. This can be done with the Vimball utility. See **|vimball|.**

It's good if you add a line to allow automatic updating. See **|glvs-plugins|.**

Add new menus

By now you know that Vim is very flexible. This includes the menus used in the GUI. You can define your own menu entries to make certain commands easily accessible. This is for mouse-happy users only.

Using filetypes

When you are editing a file of a certain type, for example a C program or a shell script, you often use the same option settings and mappings. You quickly get tired of manually setting these each time. This chapter explains how to do it automatically.

250. Plugins for a filetype

How to start using filetype plugins has already been discussed here: **|add-filetype-plugin|.** But you probably are not satisfied with the default settings, because they have been kept minimal. Suppose that for C files you want to set the 'softtabstop' option to 4 and define a mapping to insert a three-line comment. You do this with only two steps:

your-runtime-dir

1. Create your own runtime directory. On Unix this usually is "~/.vim". In this directory create the "ftplugin" directory:

mkdir ~/.vim
mkdir ~/.vim/ftplugin

When you are not on Unix, check the value of the 'runtimepath' option to see where Vim will look for the "ftplugin" directory:
set runtimepath

You would normally use the first directory name (before the first comma). You might want to prepend a directory name to the **'runtimepath'** option in your **|vimrc|** file if you don't like the default value.

2. Create the file "~/.vim/ftplugin/c.vim", with the contents:
setlocal softtabstop=4
noremap <buffer> <LocalLeader>c o/************<CR><CR>/<Esc>**

Try editing a C file. You should notice that the 'softtabstop' option is set to 4. But when you edit another file it's reset to the default zero. That is because the ":setlocal" command was used. This sets the **'softtabstop'** option only locally to the buffer. As soon as you edit another buffer, it will be set to the value set for that buffer. For a new buffer it will get the default value or the value from the last ":set" command.

Likewise, the mapping for "\c" will disappear when editing another buffer. The ":map <buffer>" command creates a mapping that is local to the current buffer. This works with any mapping command: ":map!", ":vmap", etc. The |<LocalLeader>| in the mapping is replaced with the value of the "maplocalleader" variable.

You can find examples for filetype plugins in this directory:

$VIMRUNTIME/ftplugin/

More details about writing a filetype plugin can be found here: **|write-plugin|.**

251. Adding a filetype

If you are using a type of file that is not recognized by Vim, this is how to get it recognized. You need a runtime directory of your own. See |`your-runtime-dir`| above.

Create a file "filetype.vim" which contains an autocommand for your filetype

(Autocommands were explained in section |40.3|.) Example:

```
augroup filetypedetect
au BufNewFile,BufRead *.xyz     setf xyz
augroup END
```

This will recognize all files that end in ".xyz" as the "xyz" filetype. The ":augroup" commands put this autocommand in the "filetypedetect" group. This allows removing all autocommands for filetype detection when doing ":filetype off". The "setf" command will set the 'filetype' option to its argument, unless it was set already. This will make sure that 'filetype' isn't set twice.

You can use many different patterns to match the name of your file. Directory names can also be included. See **|autocmd-patterns|.** For example, the files under "/usr/share/scripts/" are all "ruby" files, but don't have the expected file name extension. Adding this to the example above:

```
augroup filetypedetect
au BufNewFile,BufRead *.xyz                     setf xyz
au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
augroup END
```

However, if you now edit a file /usr/share/scripts/README.txt, this is not a ruby file. The danger of a pattern ending in "*" is that it quickly matches too many files. To avoid trouble with this, put the filetype.vim file in another directory, one that is at the end of 'runtimepath'. For Unix for example, you could use "~/.vim/after/filetype.vim".
   You now put the detection of text files in ~/.vim/filetype.vim:

```
augroup filetypedetect
au BufNewFile,BufRead *.txt                     setf text
augroup END
```

That file is found in 'runtimepath' first. Then use this in ~/.vim/after/filetype.vim, which is found last:

```
augroup filetypedetect
au BufNewFile,BufRead /usr/share/scripts/*      setf ruby
```

```
	augroup END
```

What will happen now is that Vim searches for "filetype.vim" files in each directory in 'runtimepath'. First ~/.vim/filetype.vim is found. The autocommand to catch *.txt files is defined there. Then Vim finds the filetype.vim file in $VIMRUNTIME, which is halfway 'runtimepath'. Finally ~/.vim/after/filetype.vim is found and the autocommand for detecting ruby files in /usr/share/scripts is added.
   When you now edit /usr/share/scripts/README.txt, the autocommands are checked in the order in which they were defined. The *.txt pattern matches, thus "setf text" is executed to set the filetype to "text". The pattern for ruby matches too, and the "setf ruby" is executed. But since 'filetype' was already set to "text", nothing happens here.
   When you edit the file /usr/share/scripts/foobar the same autocommands are checked. Only the one for ruby matches and "setf ruby" sets 'filetype' to ruby.

- **RECOGNIZING BY CONTENTS**
  If your file cannot be recognized by its file name, you might be able to recognize it by its contents. For example, many script files start with a line like:

```
	#!/bin/xyz
```

To recognize this script create a file "scripts.vim" in your runtime directory (same place where filetype.vim goes). It might look like this:

```
	if did_filetype()
	  finish
	endif
	if getline(1) =~ '^#!.*[/\\]xyz\>'
	  setf xyz
	endif
```

The first check with did_filetype() is to avoid that you will check the contents of files for which the filetype was already detected by the file name. That avoids wasting time on checking the file when the "setf" command won't do anything.
   The scripts.vim file is sourced by an autocommand in the default filetype.vim file. Therefore, the order of checks is:

```
	1. filetype.vim files before $VIMRUNTIME in 'runtimepath'
	2. first part of $VIMRUNTIME/filetype.vim
```

3. all scripts.vim files in 'runtimepath'
4. remainder of $VIMRUNTIME/filetype.vim
5. filetype.vim files after $VIMRUNTIME in 'runtimepath'

If this is not sufficient for you, add an autocommand that matches all files and sources a script or executes a function to check the contents of the file.

Your own syntax highlighted

Vim comes with highlighting for a couple of hundred different file types. If the file you are editing isn't included, read this chapter to find out how to get this type of file highlighted. Also see |:syn-define| in the reference manual.

252. Basic syntax commands

Let's start with the basic arguments. Before we start defining any new syntax, we need to clear out any old definitions:

```
:syntax clear
```

This isn't required in the final syntax file, but very useful when experimenting.

There are more simplifications in this chapter. If you are writing a syntax file to be used by others, read all the way through the end to find out the details.

- **LISTING DEFINED ITEMS**
  To check which syntax items are currently defined, use this command:

```
:syntax
```

To list the items in a specific syntax group use:

```
:syntax list {group-name}
```

- **MATCHING CASE**
  Some languages are not case sensitive, such as Pascal. Others, such as C, are case sensitive. You need to tell which type you have with the following commands:

```
:syntax case match
:syntax case ignore
```

The "match" argument means that Vim will match the case of syntax elements. Therefore, "int" differs from "Int" and "INT". If the "ignore" argument is used, the following are equivalent: "Procedure", "PROCEDURE" and "procedure".
The ":syntax case" commands can appear anywhere in a syntax file and affect the syntax definitions that follow. In most cases, you have only one ":syntax case" command in your syntax file; if you work with an unusual language that contains both case-sensitive and non-case-sensitive elements, however, you can scatter the ":syntax case" command throughout the file.

253. Keywords

The most basic syntax elements are keywords. To define a keyword, use the following form:

```
:syntax keyword {group} {keyword} ...
```

The **{group}** is the name of a syntax group. With the ":highlight" command you can assign colors to a {group}. The {keyword} argument is an actual keyword. Here are a few examples:

```
:syntax keyword xType int long char
:syntax keyword xStatement if then else endif
```

These commands cause the words "int", "long" and "char" to be highlighted one way and the words "if", "then", "else" and "endif" to be highlighted another way. Now you need to connect the x group names to standard Vim names. You do this with the following commands:

```
:highlight link xType Type
:highlight link xStatement Statement
```

This tells Vim to highlight "xType" like "Type" and "xStatement" like "Statement". See **|group-name|** for the standard names.

- **UNUSUAL KEYWORDS**

  The characters used in a keyword must be in the **'iskeyword'** option. If you use another character, the word will never match. Vim doesn't give a warning message for this.

The x language uses the '-' character in keywords. This is how it's done:

```
:setlocal iskeyword+=-
:syntax keyword xStatement when-not
```

The ":setlocal" command is used to change 'iskeyword' only for the current buffer. Still it does change the behavior of commands like "w" and "*". If that is not wanted, don't define a keyword but use a match (explained in the next section).

The x language allows for abbreviations. For example, "next" can be abbreviated to "n", "ne" or "nex". You can define them by using this command:

```
:syntax keyword xStatement n[ext]
```

This doesn't match "nextone", keywords always match whole words only.

254. Matches

Consider defining something a bit more complex. You want to match ordinary identifiers. To do this, you define a match syntax item. This one matches any word consisting of only lowercase letters:

```
:syntax match xIdentifier /\<\l\+\>/
```

Note:
Keywords overrule any other syntax item. Thus the keywords "if", "then", etc., will be keywords, as defined with the ":syntax keyword" commands above, even though they also match the pattern for xIdentifier.

The part at the end is a **pattern**, like it's used for searching. The // is used to surround the pattern (like how it's done in a ":substitute" command). You can use any other character, like a plus or a quote.

Now define a match for a comment. In the x language it is anything from # to the end of a line:

```
:syntax match xComment /#.*/
```

Since you can use any search pattern, you can highlight very complex things with a match item. See **|pattern|** for help on search patterns.

255. Regions

In the example x language, strings are enclosed in double quotation marks ("). To highlight strings you define a region. You need a region start (double quote) and a region end (double quote). The definition is as follows:

```
:syntax region xString start=/"/ end=/"/
```

The "start" and "end" directives define the patterns used to find the start and end of the region. But what about strings that look like this?

```
"A string with a double quote (\") in it"
```

This creates a problem: The double quotation marks in the middle of the string will end the region. You need to tell Vim to skip over any escaped double quotes in the string. Do this with the skip keyword:

```
:syntax region xString start=/"/ skip=/\\"/ end=/"/
```

The double backslash matches a single backslash, since the backslash is a special character in search patterns.

When to use a region instead of a match? The main difference is that a match item is a single pattern, which must match as a whole. A region starts as soon as the "start" pattern matches. Whether the "end" pattern is found or not doesn't matter. Thus when the item depends on the "end" pattern to match, you cannot use a region. Otherwise, regions are often simpler to define. And it is easier to use nested items, as is explained in the next section.

256. Nested items

Take a look at this comment:

```
%Get input  TODO: Skip white space
```

You want to highlight TODO in big yellow letters, even though it is in a comment that is highlighted blue. To let Vim know about this, you define the following syntax groups:

```
:syntax keyword xTodo TODO contained
:syntax match xComment /%.*/ contains=xTodo
```

In the first line, the "contained" argument tells Vim that this keyword can exist only inside another syntax item. The next line has "contains=xTodo". This indicates that the xTodo syntax element is inside it. The result is that the comment line as a whole is matched with "xComment" and made blue. The word TODO inside it is matched by xTodo and highlighted yellow (highlighting for xTodo was setup for this).

- **RECURSIVE NESTING**

  The x language defines code blocks in curly braces. And a code block may contain other code blocks. This can be defined this way:

  ```
  :syntax region xBlock start=/{/ end=/}/ contains=xBlock
  ```

  Suppose you have this text:

  ```
  while i < b {
          if a {
                  b = c;
          }
  }
  ```

  First a xBlock starts at the **{** in the first line. In the second line another { is found. Since we are inside a xBlock item, and it contains itself, a nested xBlock item will start here. Thus the "b = c" line is inside the second level xBlock region. Then a } is found in the next line, which matches with the end pattern of the region. This ends the nested xBlock. Because the } is included in the nested region, it is hidden from the first xBlock region. Then at the last } the first xBlock region ends.

- **KEEPING THE END**

  Consider the following two syntax items:

  ```
  :syntax region xComment start=/%/ end=/$/ contained
  :syntax region xPreProc start=/#/ end=/$/ contains=xComment
  ```

  You define a comment as anything from % to the end of the line. A preprocessor directive is anything from # to the end of the line. Because you can have a comment on a preprocessor line, the preprocessor definition includes a "contains=xComment" argument.

Now look what happens with this text:

```
#define X = Y  % Comment text
int foo = 1;
```

What you see is that the second line is also highlighted as xPreProc. The preprocessor directive should end at the end of the line. That is why you have used "end=/$/". So what is going wrong?

The problem is the contained comment. The comment starts with % and ends at the end of the line. After the comment ends, the preprocessor syntax continues. This is after the end of the line has been seen, so the next line is included as well.

To avoid this problem and to avoid a contained syntax item eating a needed end of line, use the "keepend" argument. This takes care of the double end-of-line matching:

```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment keepend
```

- **CONTAINING MANY ITEMS**

You can use the contains argument to specify that everything can be contained. For example:

```
:syntax region xList start=/\[/ end=/\]/ contains=ALL
```

All syntax items will be contained in this one. It also contains itself, but not at the same position (that would cause an endless loop).

You can specify that some groups are not contained. Thus contain all groups but the ones that are listed:

```
:syntax region xList start=/\[/ end=/\]/ contains=ALLBUT,xString
```

With the "TOP" item you can include all items that don't have a "contained" argument. "CONTAINED" is used to only include items with a "contained" argument. See **|:syn-contains|** for the details.

257. Following groups

The x language has statements in this form:

```
    if (condition) then
```

You want to highlight the three items differently. But "(condition)" and "then" might also appear in other places, where they get different highlighting. This is how you can do this:

```
    :syntax match xIf /if/ nextgroup=xIfCondition skipwhite
    :syntax match xIfCondition /([^)]*)/ contained nextgroup=xThen skipwhite
    :syntax match xThen /then/ contained
```

The **"nextgroup"** argument specifies which item can come next. This is not required. If none of the items that are specified are found, nothing happens. For example, in this text:

```
    if not (condition) then
```

The "if" is matched by xIf. "not" doesn't match the specified nextgroup xIfCondition, thus only the "if" is highlighted.

The **"skipwhite"** argument tells Vim that white space (spaces and tabs) may appear in between the items. Similar arguments are **"skipnl"**, which allows a line break in between the items, and "**skipempty**", which allows empty lines. Notice that "skipnl" doesn't skip an empty line, something must match after the line break.

258. Other arguments

- **MATCHGROUP**
  When you define a region, the entire region is highlighted according to the group name specified. To highlight the text enclosed in parentheses () with the group xInside, for example, use the following command:

```
    :syntax region xInside start=/(/ end=/)/
```

Suppose, that you want to highlight the parentheses differently. You can do this with a lot of convoluted region statements, or you can use the "matchgroup" argument. This tells Vim to highlight the start and end of a region with a different highlight group (in this case, the xParen group):

```
    :syntax region xInside matchgroup=xParen start=/(/ end=/)/
```

The "matchgroup" argument applies to the start or end match that comes after it. In the previous example both start and end are highlighted with xParen. To highlight the end with xParenEnd:

```
:syntax region xInside matchgroup=xParen start=/(/
        \ matchgroup=xParenEnd end=/)/
```

A side effect of using "matchgroup" is that contained items will not match in the start or end of the region. The example for "transparent" uses this.

- **TRANSPARENT**

In a C language file you would like to highlight the () text after a "while" differently from the () text after a "for". In both of these there can be nested () items, which should be highlighted in the same way. You must make sure the () highlighting stops at the matching). This is one way to do this:

```
:syntax region cWhile matchgroup=cWhile start=/while\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cFor matchgroup=cFor start=/for\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cCondNest start=/(/ end=/)/ contained transparent
```

Now you can give cWhile and cFor different highlighting. The cCondNest item can appear in either of them, but take over the highlighting of the item it is contained in. The "transparent" argument causes this.

Notice that the "matchgroup" argument has the same group as the item itself. Why define it then? Well, the side effect of using a matchgroup is that contained items are not found in the match with the start item then. This avoids that the cCondNest group matches the (just after the "while" or "for". If this would happen, it would span the whole text until the matching) and the region would continue after it. Now cCondNest only matches after the match with the start pattern, thus after the first (.

- **OFFSETS**

Suppose you want to define a region for the text between (and) after an "if". But you don't want to include the "if" or the (and). You can do this by specifying offsets for the patterns. Example:

```
:syntax region xCond start=/if\s*(/ms=e+1 end=/)/me=s-1
```

The offset for the start pattern is "**ms=e+1**". "ms" stands for Match Start. This defines an offset for the start of the match. Normally the match starts where the pattern matches. **"e+1"** means that the match now starts at the end of the pattern match, and then one character further.

The offset for the end pattern is **"me=s-1"**. "me" stands for Match End. "s-1" means the start of the pattern match and then one character back. The result is that in this text:

```
if (foo == bar)
```

Only the text "foo == bar" will be highlighted as xCond.
More about offsets here: **|:syn-pattern-offset|.**

- **ONELINE**
  The "oneline" argument indicates that the region does not cross a line boundary. For example:

  ```
  :syntax region xIfThen start=/if/ end=/then/ oneline
  ```

  This defines a region that starts at "if" and ends at "then". But if there is no "then" after the "if", the region doesn't match.

  Note:
  When using "oneline" the region doesn't start if the end pattern doesn't match in the same line. Without "oneline" Vim does _not_ check if there is a match for the end pattern. The region starts even when the end pattern doesn't match in the rest of the file.

- **CONTINUATION LINES AND AVOIDING THEM**
  Things now become a little more complex. Let's define a preprocessor line. This starts with a # in the first column and continues until the end of the line. A line that ends with \ makes the next line a continuation line. The way you handle this is to allow the syntax item to contain a continuation pattern:

  ```
  :syntax region xPreProc start=/^#/ end=/$/ contains=xLineContinue
  :syntax match xLineContinue "\\$" contained
  ```

  In this case, although xPreProc normally matches a single line, the group contained in it (namely xLineContinue) lets it go on for more than one line.

For example, it would match both of these lines:

```
#define SPAM  spam spam spam \
          bacon and spam
```

In this case, this is what you want. If it is not what you want, you can call for the region to be on a single line by adding "excludenl" to the contained pattern. For example, you want to highlight "end" in xPreProc, but only at the end of the line. To avoid making the xPreProc continue on the next line, like xLineContinue does, use "excludenl" like this:

```
:syntax region xPreProc start=/^#/ end=/$/
     \ contains=xLineContinue,xPreProcEnd
:syntax match xPreProcEnd excludenl /end$/ contained
:syntax match xLineContinue "\\$" contained
```

"excludenl" must be placed before the pattern. Since "xLineContinue" doesn't have "excludenl", a match with it will extend xPreProc to the next line as before.

259. Clusters

One of the things you will notice as you start to write a syntax file is that you wind up generating a lot of syntax groups. Vim enables you to define a collection of syntax groups called a cluster.

Suppose you have a language that contains for loops, if statements, while loops, and functions. Each of them contains the same syntax elements: numbers and identifiers. You define them like this:

```
:syntax match xFor /^for.*/ contains=xNumber,xIdent
:syntax match xIf /^if.*/ contains=xNumber,xIdent
:syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

You have to repeat the same "contains=" every time. If you want to add another contained item, you have to add it three times. Syntax clusters simplify these definitions by enabling you to have one cluster stand for several syntax groups.

To define a cluster for the two items that the three groups contain, use the following command:

```
:syntax cluster xState contains=xNumber,xIdent
```

Clusters are used inside other syntax items just like any syntax group. Their names start with @. Thus, you can define the three groups like this:

```
:syntax match xFor /^for.*/ contains=@xState
:syntax match xIf /^if.*/ contains=@xState
:syntax match xWhile /^while.*/ contains=@xState
```

You can add new group names to this cluster with the "add" argument:

```
:syntax cluster xState add=xString
```

You can remove syntax groups from this list as well:

```
:syntax cluster xState remove=xNumber
```

260. Including another syntax file

The C++ language syntax is a superset of the C language. Because you do not want to write two syntax files, you can have the C++ syntax file read in the one for C by using the following command:

```
:runtime! syntax/c.vim
```

The ":runtime!" command searches 'runtimepath' for all "syntax/c.vim" files. This makes the C parts of the C++ syntax be defined like for C files. If you have replaced the c.vim syntax file, or added items with an extra file, these will be loaded as well.

After loading the C syntax items the specific C++ items can be defined. For example, add keywords that are not used in C:

```
:syntax keyword cppStatement    new delete this friend using
```

This works just like in any other syntax file.

Now consider the Perl language. A Perl script consists of two distinct parts: a documentation section in POD format, and a program written in Perl itself. The POD section starts with "=head" and ends with "=cut".

You want to define the POD syntax in one file, and use it from the Perl syntax file. The ":syntax include" command reads in a syntax file and stores the elements it defined in a syntax cluster. For Perl, the statements are as follows:

```
:syntax include @Pod <sfile>:p:h/pod.vim
:syntax region perlPOD start=/^=head/ end=/^=cut/ contains=@Pod
```

When "=head" is found in a Perl file, the perlPOD region starts. In this region the @Pod cluster is contained. All the items defined as top-level items in the pod.vim syntax files will match here. When "=cut" is found, the region ends and we go back to the items defined in the Perl file.

The ":syntax include" command is clever enough to ignore a ":syntax clear" command in the included file. And an argument such as "contains=ALL" will only contain items defined in the included file, not in the file that includes it.

The "<sfile>:p:h/" part uses the name of the current file (<sfile>), expands it to a full path (:p) and then takes the head (:h). This results in the directory name of the file. This causes the pod.vim file in the same directory to be included.

261. Synchronizing

Compilers have it easy. They start at the beginning of a file and parse it straight through. Vim does not have it so easy. It must start in the middle, where the editing is being done. So how does it tell where it is?

The secret is the ":syntax sync" command. This tells Vim how to figure out where it is. For example, the following command tells Vim to scan backward for the beginning or end of a C-style comment and begin syntax coloring from there:

```
:syntax sync ccomment
```

You can tune this processing with some arguments. The "minlines" argument tells Vim the minimum number of lines to look backward, and "maxlines" tells the editor the maximum number of lines to scan.

For example, the following command tells Vim to look at least 10 lines before the top of the screen:

```
:syntax sync ccomment minlines=10 maxlines=500
```

If it cannot figure out where it is in that space, it starts looking farther and farther back until it figures out what to do. But it looks no farther back than 500 lines. (A large "maxlines" slows down processing. A small one might cause synchronization to fail.)

To make synchronizing go a bit faster, tell Vim which syntax items can be skipped. Every match and region that only needs to be used when actually displaying text can be given the "display" argument.

By default, the comment to be found will be colored as part of the Comment syntax group. If you want to color things another way, you can specify a different syntax group:

```
:syntax sync ccomment xAltComment
```

If your programming language does not have C-style comments in it, you can try another method of synchronization. The simplest way is to tell Vim to space back a number of lines and try to figure out things from there. The following command tells Vim to go back 150 lines and start parsing from there:

```
:syntax sync minlines=150
```

A large "minlines" value can make Vim slower, especially when scrolling backwards in the file.

Finally, you can specify a syntax group to look for by using this command:

```
:syntax sync match {sync-group-name}
        \ grouphere {group-name} {pattern}
```

This tells Vim that when it sees {pattern} the syntax group named {group-name} begins just after the pattern given. The {sync-group-name} is used to give a name to this synchronization specification. For example, the sh scripting language begins an if statement with "if" and ends it with "fi":

```
if [ --f file.txt ] ; then
        echo "File exists"
fi
```

To define a "grouphere" directive for this syntax, you use the following command:

```
:syntax sync match shIfSync grouphere shIf "\<if\>"
```

The "groupthere" argument tells Vim that the pattern ends a group. For example, the end of the if/fi group is as follows:

```
:syntax sync match shIfSync groupthere NONE "\<fi\>"
```

In this example, the NONE tells Vim that you are not in any special syntax region. In particular, you are not inside an if block.

You also can define matches and regions that are with no "grouphere" or "groupthere" arguments. These groups are for syntax groups skipped during synchronization. For example, the following skips over anything inside {}, even if it would normally match another synchronization method:

```
:syntax sync match xSpecial /{.*}/
```

More about synchronizing in the reference manual: |:syn-sync|.

262. Installing a syntax file

When your new syntax file is ready to be used, drop it in a "syntax" directory in 'runtimepath'. For Unix that would be "~/.vim/syntax".
The name of the syntax file must be equal to the file type, with ".vim" added. Thus for the x language, the full path of the file would be:

```
~/.vim/syntax/x.vim
```

You must also make the file type be recognized. See |43.2|.

If your file works well, you might want to make it available to other Vim users. First read the next section to make sure your file works well for others. Then e-mail it to the Vim maintainer: <maintainer@vim.org>. Also explain how the filetype can be detected. With a bit of luck your file will be included in the next Vim version!

- **ADDING TO AN EXISTING SYNTAX FILE**

We were assuming you were adding a completely new syntax file. When an existing syntax file works, but is missing some items, you can add items in a separate file. That avoids changing the distributed syntax file, which will be lost when installing a new version of Vim.
Write syntax commands in your file, possibly using group names from the existing syntax. For example, to add new variable types to the C syntax file:

```
:syntax keyword cType off_t uint
```

Write the file with the same name as the original syntax file. In this case "c.vim". Place it in a directory near the end of 'runtimepath'. This makes it loaded after the original syntax file. For Unix this would be:

```
~/.vim/after/syntax/c.vim
```

263. Portable syntax file layout

Wouldn't it be nice if all Vim users exchange syntax files? To make this possible, the syntax file must follow a few guidelines.

Start with a header that explains what the syntax file is for, who maintains it and when it was last updated. Don't include too much information about changes history, not many people will read it. Example:

```
" Vim syntax file
" Language:     C
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" Last Change:  2001 Jun 18
" Remark:       Included by the C++ syntax.
```

Use the same layout as the other syntax files. Using an existing syntax file as an example will save you a lot of time.

- Choose a good, descriptive name for your syntax file. Use lowercase letters and digits. Don't make it too long, it is used in many places: The name of the syntax file "name.vim", 'filetype', b:current_syntax and the start of each syntax group (nameType, nameStatement, nameString, etc).

- Start with a check for "b:current_syntax". If it is defined, some other syntax file, earlier in 'runtimepath' was already loaded:

```
if exists("b:current_syntax")
  finish
endif
```

To be compatible with Vim 5.8 use:

```
if version < 600
  syntax clear
elseif exists("b:current_syntax")
  finish
endif
```

- Set "b:current_syntax" to the name of the syntax at the end. Don't forget that included files do this too, you might have to reset "b:current_syntax" if you include two files.

  If you want your syntax file to work with Vim 5.x, add a check for v:version. See yacc.vim for an example.

  Do not include anything that is a user preference. Don't set 'tabstop', 'expandtab', etc. These belong in a filetype plugin.

  Do not include mappings or abbreviations. Only include setting 'iskeyword' if it is really necessary for recognizing keywords.

  To allow users select their own preferred colors, make a different group name for every kind of highlighted item. Then link each of them to one of the standard highlight groups. That will make it work with every color scheme. If you select specific colors it will look bad with some color schemes. And don't forget that some people use a different background color, or have only eight colors available.

  For the linking use "hi def link", so that the user can select different highlighting before your syntax file is loaded. Example:

  ```
  hi def link nameString      String
  hi def link nameNumber      Number
  hi def link nameCommand     Statement
  ... etc ...
  ```

  Add the "display" argument to items that are not used when syncing, to speed up scrolling backwards and CTRL-L.

Select your language

The messages in Vim can be given in several languages. This chapter explains how to change which one is used. Also, the different ways to work with files in various languages is explained.

264. Language for Messages

To see what the current language is, use this command:

```
:language
```

Note:

Using different languages only works when Vim was compiled to handle it. To find out if it works, use the ":version" command and check the output for "+gettext" and "+multi_lang". If they are there, you are OK. If you see "-gettext" or "-multi_lang" you will have to find another Vim.

The first way is to set the environment to the desired language before starting Vim. Example for Unix:

```
env LANG=de_DE.ISO_8859-1  vim
```

If you want to change language while Vim is running, you can use the second method:

```
:language fr_FR.ISO_8859-1
```

• Do-it-yourself message translation

If translated messages are not available for your language, you could write them yourself. To do this, get the source code for Vim and the GNU gettext package. After unpacking the sources, instructions can be found in the directory src/po/README.txt.

It's not too difficult to do the translation. You don't need to be a programmer. You must know both English and the language you are translating to, of course.

When you are satisfied with the translation, consider making it available to others. Upload it at vim-online http://vim.sf.net or e-mail it to the Vim maintainer <maintainer@vim.org>. Or both.

265. Using another encoding

Vim guesses that the files you are going to edit are encoded for your language. For many European languages this is "latin1". Then each byte is one character. That means there are 256 different characters possible. For Asian languages this is not sufficient. These mostly use a double-byte encoding, providing for over ten thousand possible characters. This still isn't enough when a text is to contain several different languages. This is where Unicode comes in. It was designed to include all characters used in commonly used languages. This is the "Super encoding that replaces all others". But it isn't used that much yet.

Fortunately, Vim supports these three kinds of encodings. And, with some restrictions, you can use them even when your environment uses another language than the text.

Nevertheless, when you only edit files that are in the encoding of your language, the default should work fine and you don't need to do anything. The following is only relevant when you want to edit different languages.

Note:

Using different encodings only works when Vim was compiled to handle it. To find out if it works, use the ":version" command and check the output for "+multi_byte". If it's there, you are OK. If you see "-multi_byte" you will have to find another Vim.

- **USING UNICODE IN THE GUI**
  :set encoding=utf-8

266. Editing files with a different encoding

267. Entering language text

If all other methods fail, you can enter any character with **CTRL-V**:

encoding	**type**	**range**
8-bit	CTRL-V 123	decimal 0-255
8-bit	CTRL-V x a1	hexadecimal 00-ff
16-bit	CTRL-V u 013b	hexadecimal 0000-ffff
31-bit	CTRL-V U 001303a4	hexadecimal 00000000-7fffffff

268. a

269. a

270. a