Vim User Manual

Bram Moolenaar

Contents

00. Introduction to Vim

Introduction

Vim stands for Vi IMproved. It used to be Vi IMitation, but there are so many improvements that a name change was appropriate. Vim is a text editor which includes almost all the commands from the Unix program "Vi" and a lot of new ones. It is very useful for editing programs and other plain text. All commands are given with the keyboard. This has the advantage that you can keep your fingers on the keyboard and your eyes on the screen. For those who want it, there is mouse support and a GUI version with scrollbars and menus (see |`:h gui.txt`|).

An overview of this manual can be found in the file "help.txt", |`:h help.txt`|. It can be accessed from within Vim with the <Help> or <F1> key and with the |`:help`| command (just type "`:help`", without the bars or quotes). The `'helpfile'` option can be set to the name of the help file, in case it is not located in the default place.

Throughout this manual the differences between Vi and Vim are mentioned in curly braces, like this: {Vi does not have on-line help}. See |`:h vi_diff.txt`| for a summary of the differences between Vim and Vi.

This manual refers to Vim on various machines. There may be small differences between different computers and terminals. Besides the remarks given in this document, there is a separate document for each supported system, see |`:h sys-file-list`|.

Vim is pronounced as one word, like Jim, not vi-ai-em. It's written with a capital, since it's a name, again like Jim.

This manual is a reference for all the Vim commands and options. This is not an introduction to the use of Vi or Vim, it gets a bit complicated here and there. For beginners, there is a hands-on |`tutor`|. To learn using Vim, read the user manual 1.

There are many books on Vi that contain a section for beginners. There are two books I can recommend:

- *Vim - Vi Improved* by Steve Oualline
  This is the very first book completely dedicated to Vim. It is very good for beginners. The most often used commands are explained with pictures and examples. The less often used commands are also explained, the more advanced features are summarized. There is a comprehensive index and a quick reference. Parts of this book have been included in the user manual |`frombook`|. Published by New Riders Publishing. ISBN: 0735710015 For more information try one of these:
  - http://iccf-holland.org/click5.html
  - http://www.vim.org/iccf/click5.html
- *Learning the Vi editor* by Linda Lamb and Arnold Robbins
  This is a book about Vi that includes a chapter on Vim (in the sixth edition). The first steps in Vi are explained very well. The commands that Vim adds are only briefly mentioned. There is also a German translation. Published by O'Reilly. ISBN: 1-56592-426-6.

Vim on the internet

The Vim pages contain the most recent information about Vim. They also contain links to the most recent version of Vim. The FAQ is a list of Frequently Asked Questions. Read this if you have problems.

- VIM home page: http://www.vim.org/
- VIM FAQ: http://vimdoc.sf.net/

- Downloading: ftp://ftp.vim.org/pub/vim/MIRRORS

Usenet News group where Vim is discussed:
`comp.editors`
This group is also for other editors. If you write about Vim, don't forget to mention that.

There are several mailing lists for Vim:

- `vim@vim.org` For discussions about using existing versions of Vim: Useful mappings, questions, answers, where to get a specific version, etc. There are quite a few people watching this list and answering questions, also for beginners. Don't hesitate to ask your question here.
- `vim-dev@vim.org` For discussions about changing Vim: New features, porting, patches, beta-test versions, etc.
- `vim-announce@vim.org` Announcements about new versions of Vim; also for beta-test versions and ports to different systems. This is a read-only list.
- `vim-multibyte@vim.org` For discussions about using and improving the multi-byte aspects of Vim.
- `vim-mac@vim.org` For discussions about using and improving the Macintosh version of Vim.
- See http://www.vim.org/maillist.php for the latest information.

NOTE:

- You can only send messages to these lists if you have subscribed!
- You need to send the messages from the same location as where you subscribed from (to avoid spam mail).
- Maximum message size is 40000 characters.

If you want to join, send a message to <vim-subscribe@vim.org>. Make sure that your "From:" address is correct. Then the list server will give you help on how to subscribe.

For more information and archives look on the Vim maillist page: http://www.vim.org/maillist.php.

Bug reports:

Send bug reports to: Vim bugs `bugs@vim.org`. This is not a maillist but the message is redirected to the Vim maintainer. Please be brief; all the time that is spent on answering mail is subtracted from the time that is spent on improving Vim! Always give a reproducible example and try to find out which settings or other things influence the appearance of the bug. Try different machines, if possible. Send me patches if you can!

It will help to include information about the version of Vim you are using and your setup. You can get the information with this command:

```
:so $VIMRUNTIME/bugreport.vim
```

This will create a file "bugreport.txt" in the current directory, with a lot of information of your environment. Before sending this out, check if it doesn't contain any confidential information!

If Vim crashes, please try to find out where. You can find help on this here: |`:h debug.txt`|.

In case of doubt or when you wonder if the problem has already been fixed but you can't find a fix for it, become a member of the vim-dev maillist and ask your question there: |`maillist`|

Since Vim internally doesn't use dates for editing, there is no year 2000 problem to worry about. Vim does use the time in the form of seconds since January 1st 1970. It is used for a time-stamp check of the edited file and the swap file, which is not critical and should only cause warning messages.

There might be a year 2038 problem, when the seconds don't fit in a 32 bit int anymore. This depends on the compiler, libraries and operating system. Specifically, `time_t` and the `ctime()` function are used. And the `time_t` is stored in four bytes in the swap file. But that's only used for printing a file date/time for recovery, it will never affect normal editing.

The Vim `strftime()` function directly uses the `strftime()` system function. `localtime()` uses the `time()` system function. `getftime()` uses the time returned by the `stat()` system function. If your system libraries are year 2000 compliant, Vim is too.

The user may create scripts for Vim that use external commands. These might introduce Y2K problems, but those are not really part of Vim itself.

Credits

Most of Vim was written by Bram Moolenaar <Bram@vim.org>.

Parts of the documentation come from several Vi manuals, written by:

- W.N. Joy
- Alan P.W. Hewett
- Mark Horton

The Vim editor is based on Stevie and includes (ideas from) other software, worked on by the people mentioned here. Other people helped by sending me patches, suggestions and giving feedback about what is good and bad in Vim.

Vim would never have become what it is now, without the help of these people!

Ron Aaron	Win32 GUI changes
Mohsin Ahmed	encryption
Zoltan Arpadffy	work on VMS port
Tony Andrews	Stevie
Gert van Antwerpen	changes for DJGPP on MS-DOS
Berkeley DB(3)	ideas for swap file implementation
Keith Bostic	Nvi
Walter Briscoe	Makefile updates, various patches
Ralf Brown	SPAWNO library for MS-DOS
Robert Colon	many useful remarks
Marcin Dalecki	GTK+ GUI port, toolbar icons, gettext()
Kayhan Demirel	sent me news in Uganda
Chris & John Downey	xvi (ideas for multi-windows version)
Henk Elbers	first VMS port
Daniel Elstner	GTK+ 2 port
Eric Fischer	Mac port, `'cindent'`, and other improvements
Benji Fisher	Answering lots of user questions
Bill Foster	Athena GUI port
Google	Lets me work on Vim one day a week
Loic Grenie	xvim (ideas for multi windows version)
Sven Guckes	Vim promoter and previous WWW page maintainer

Darren Hiebert	Exuberant ctags
Jason Hildebrand	GTK+ 2 port
Bruce Hunsaker	improvements for VMS port
Andy Kahn	Cscope support, GTK+ GUI port
Oezguer Kesim	Maintainer of Vim Mailing Lists
Axel Kielhorn	work on the Macintosh port
Steve Kirkendall	Elvis
Roger Knobbe	original port to Windows NT
Sergey Laskavy	Vim's help from Moscow
Felix von Leitner	Previous maintainer of Vim Mailing Lists
David Leonard	Port of Python extensions to Unix
Avner Lottem	Edit in right-to-left windows
Flemming Madsen	X11 client-server, various features and patches
Tony Mechelynck	answers many user questions
Paul Moore	Python interface extensions, many patches
Katsuhito Nagano	Work on multi-byte versions
Sung-Hyun Nam	Work on multi-byte versions
Vince Negri	Win32 GUI and generic console enhancements
Steve Oualline	Author of the first Vim book |`frombook`|
Dominique Pelle	valgrind reports and many fixes
A.Politz	Many bug reports and some fixes
George V. Reilly	Win32 port, Win32 GUI start-off
Stephen Riehm	bug collector
Stefan Roemer	various patches and help to users
Ralf Schandl	IBM OS & 390 port
Olaf Seibert	DICE and BeBox version, regexp improvements
Mortaza Shiran	Farsi patches
Peter da Silva	termlib
Paul Slootman	OS/2 port
Henry Spencer	regular expressions
Dany St-Amant	Macintosh port
Tim Thompson	Stevie
G. R. (Fred) Walter	Stevie
Sven Verdoolaege	Perl interface
Robert Webb	Command-line completion, GUI versions, and lots of patches
Ingo Wilken	Tcl interface
Mike Williams	PostScript printing
Juergen Weigert	Lattice version, AUX improvements, UNIX and MS-DOS ports, autoconf
Stefan 'Sec' Zehl	Maintainer of vim.org

I wish to thank all the people that sent me bug reports and suggestions. The list is too long to mention them all here. Vim would not be the same without the ideas from all these people: They keep Vim alive!

In this documentation there are several references to other versions of Vi:

Vi

Vi "the original". Without further remarks this is the version of Vi that appeared in Sun OS 4.x. `:version` returns "Version 3.7, 6/7/85". Sometimes other versions are referred to. Only runs under Unix. Source code only available with a license. More information on Vi can be found through: http://vi-editor.org [doesn't currently work...].

Posix

From the IEEE standard 1003.2, Part 2: Shell and utilities. Generally known as "Posix". This is a textual description of how Vi is supposed to work. See |`:h posix-compliance`|.

Nvi

The "New" Vi. The version of Vi that comes with BSD 4.4 and FreeBSD. Very good compatibility with the original Vi, with a few extensions. The version used is 1.79. `:version` returns "Version 1.79 (10/23/96)". There has been no release the last few years, although there is a development version 1.81. Source code is freely available.

Elvis

Another Vi clone, made by Steve Kirkendall. Very compact but isn't as flexible as Vim.* The version used is 2.1. It is still being developed. Source code is freely available.

Notation

When syntax highlighting is used to read this, text that is not typed literally is often highlighted with the Special group. These are items in `[]`, `{}` and `<>`, and `CTRL-X`.

Note that Vim uses all possible characters in commands. Sometimes the `[]`, `{}` and `<>` are part of what you type, the context should make this clear.

[ ] Characters in square brackets are optional.

[count] An optional number that may precede the command to multiply or iterate the command. If no number is given, a count of one is used, unless otherwise noted. Note that in this manual the [count] is not mentioned in the description of the command, but only in the explanation. This was done to make the commands easier to look up. If the `'showcmd'` option is on, the (partially) entered count is shown at the bottom of the window. You can use <Del> to erase the last digit (|`:h N<Del>`|).

["x] See |`:h registers`|. An optional register designation where text can be stored. The x is a single character between 'a' and 'z' or 'A' and 'Z' or '"", and in some cases (with the put command) between '0' and '9', '%', '#', or others. The uppercase and lowercase letter designate the same register, but the lowercase letter is used to overwrite the previous register contents, while the uppercase letter is used to append to the previous register contents. Without the ""x" or with """" the stored text is put into the unnamed register.

{} Curly braces denote parts of the command which must appear, but which can take a number of different values. The differences between Vim and Vi are also given in curly braces (this will be clear from the context).

{char1-char2} A single character from the range char1 to char2. For example: `{a-z}` is a lowercase letter. Multiple ranges may be concatenated. For example, `{a-zA-Z0-9}` is any alphanumeric character.

{motion} A command that moves the cursor. These are explained in |`:h motion.txt`|. Examples:

`w`	to start of next word
`b`	to begin of current word
`4j`	four lines down
`/The<CR>`	to next occurrence of "The"

This is used after an |`:h operator`| command to move over the text that is to be operated upon.

- If the motion includes a count and the operator also has a count, the two counts are multiplied. For example: "2d3w" deletes six words.
- The motion can be backwards, e.g. "db" to delete to the start of the word.
- The motion can also be a mouse click. The mouse is not supported in every terminal though.
- The ":omap" command can be used to map characters while an operator is pending.
- Ex commands can be used to move the cursor. This can be used to call a function that does some complicated motion. The motion is always characterwise exclusive, no matter what ":" command is used. This means it's impossible to include the last character of a line without the line break (unless `'virtualedit'` is set). If the Ex command changes the text before where the operator starts or jumps to another buffer the result is unpredictable. It is possible to change the text further down. Jumping to another buffer is possible if the current buffer is not unloaded.

{Visual} A selected text area. It is started with the "v", "V", or CTRL-V command, then any cursor movement command can be used to change the end of the selected text. This is used before an |`:h operator`| command to highlight the text that is to be operated upon. See |`:h Visual-mode`|.

<character> A special character from the table below, optionally with modifiers, or a single ASCII character with modifiers.

'c' A single ASCII character.

CTRL-{char} `{char}` typed as a control character; that is, typing `{char}` while holding the CTRL key down. The case of `{char}` does not matter; thus CTRL-A and CTRL-a are equivalent. But on some terminals, using the SHIFT key will produce another code, don't use it then.

`'option'` An option, or parameter, that can be set to a value, is enclosed in single quotes. See |`:h options`|.

"command" A reference to a command that you can type is enclosed in double quotes.

These names for keys are used in the documentation. They can also be used with the "`:map`" command (insert the key name by pressing CTRL-K and then the key you want the name for).

notation	meaning	equivalent	decimal	value(s)
<Nul>	zero	CTRL-@	0 (stored as 10)	
<BS>	backspace	CTRL-H	8	
<Tab>	tab	CTRL-I	9	
<NL>	linefeed	CTRL-J	10 (used for <Nul>)	
<FF>	formfeed	CTRL-L	12	
<CR>	carriage return	CTRL-M	13	
<Return>	same as <CR>			
<Enter>	same as <CR>			
<Esc>	escape	CTRL-[	27	
<Space>	space		32	
<lt>	less-than	<	60	
<Bslash>	backslash		92	
<Bar>	vertical bar	|	124	
<Del>	delete		127	
<CSI>	command sequence intro	ALT-Esc	155	
<xCSI>	CSI when typed in the GUI			
<EOL>	end-of-line (can be <CR>, <LF> or <CR><LF>, depends on system and `'fileformat'`)			
<Up>	cursor-up			
<Down>	cursor-down			
<Left>	cursor-left			
<Right>	cursor-right			
<S-Up>	shift-cursor-up			
<S-Down>	shift-cursor-down			
<S-Left>	shift-cursor-left			
<S-Right>	shift-cursor-right			
<C-Left>	control-cursor-left			
<C-Right>	control-cursor-right			
<F1> - <F12>	function keys 1 to 12			
<S-F1> - <S-F12>	shift-function keys 1 to 12			
<Help>	help key			
<Undo>	undo key			
<Insert>	insert key			
<Home>	home			
<End>	end			
<PageUp>	page-up			
<PageDown>	page-down			
<kHome>	keypad home (upper left)			
<kEnd>	keypad end (lower left)			
<kPageUp>	keypad page-up (upper right)			
<kPageDown>	keypad page-down (lower right)			
<kPlus>	keypad +			
<kMinus>	keypad -			
<kMultiply>	keypad *			
<kDivide>	keypad /			
<kEnter>	keypad Enter			
<kPoint>	keypad Decimal point			
<k0> - <k9>	keypad 0 to 9			
<S-...>	shift-key			
<C-...>	control-key			
<M-...>	alt-key or meta-key			
<A-...>	same as <M-...>			
<D-...>	command-key (Macintosh only)			
<t_xx>	key with "xx" entry in termcap			

Note: The shifted cursor keys, the help key, and the undo key are only available on a few terminals. On the Amiga, shifted function key 10 produces a code (CSI) that is also used by key sequences. It will be recognized only after typing another key.

Note: There are two codes for the delete key. 127 is the decimal ASCII value for the delete key, which is always recognized. Some delete keys send another value, in which case this value is obtained from the termcap entry "kD". Both values have the same effect. Also see |`:h :fixdel`|.

Note: The keypad keys are used in the same way as the corresponding "normal" keys. For example, <kHome> has the same effect as <Home>. If a keypad key sends the same raw key code as its non-keypad equivalent, it will be recognized as the non-keypad code. For example, when <kHome> sends the same code as <Home>, when pressing <kHome> Vim will think <Home> was pressed. Mapping <kHome> will not work then.

Examples are often given in the <> notation. Sometimes this is just to make clear what you need to type, but often it can be typed literally, e.g., with the "`:map`" command. The rules are:

1. Any printable characters are typed directly, except backslash and '<'
2. A backslash is represented with "\\", double backslash, or "<Bslash>".
3. A real '<' is represented with "\<" or "<lt>". When there is no confusion possible, a '<' can be used directly.
4. "<key>" means the special key typed. This is the notation explained in the table above. A few examples:

<Esc>	Escape key
<C-G>	CTRL-G
<Up>	cursor up key
<C-LeftMouse>	Control- left mouse click
<S-F11>	Shifted function key 11
<M-a>	Meta- a ('a' with bit 8 set)
<M-A>	Meta- A ('A' with bit 8 set)
<t_kd>	"kd" termcap entry (cursor down key)

If you want to use the full <> notation in Vim, you have to make sure the '<' flag is excluded from `'cpoptions'` (when `'compatible'` is not set, it already is by default).

```
:set cpo-=<
```

The <> notation uses <lt> to escape the special meaning of key names. Using a backslash also works, but only when `'cpoptions'` does not include the `'B'` flag.

Examples for mapping CTRL-H to the six characters "<Home>":

```
:imap <C-H> \<Home>
:imap <C-H> <lt>Home>
```

The first one only works when the `'B'` flag is not in `'cpoptions'`. The second one always works. To get a literal "<lt>" in a mapping:

```
:map <C-L> <lt>lt>
```

For mapping, abbreviation and menu commands you can then copy-paste the examples and use them directly. Or type them literally, including the '<' and '>' characters. This does NOT work for other commands, like "`:set`" and "`:autocmd`"!

Modes, introduction

Vim has six BASIC modes:

Normal Mode

In Normal mode you can enter all the normal editor commands. If you start the editor you are in this mode (unless you have set the `'insertmode'` option, see below). This is also known as command mode.

Visual mode

This is like Normal mode, but the movement commands extend a highlighted area. When a non-movement command is used, it is executed for the highlighted area. See |`:h Visual-mode`|. If the `'showmode'` option is on `-- VISUAL --` is shown at the bottom of the window.

Select mode

This looks most like the MS-Windows selection mode. Typing a printable character deletes the selection and starts Insert mode. See |`:h Select-mode`|. If the `'showmode'` option is on `-- SELECT --` is shown at the bottom of the window.

Insert mode

In Insert mode the text you type is inserted into the buffer. See |`:h Insert-mode`|. If the `'showmode'` option is on `-- INSERT --` is shown at the bottom of the window.

Command-line mode, Cmdlinemode

In Command-line mode (also called Cmdline mode) you can enter one line of text at the bottom of the window. This is for the Ex commands, ":", the pattern search commands, "?" and "/", and the filter command, "!". |`:h Cmdline-mode`|

Ex mode

Like Command-line mode, but after entering a command you remain in Ex mode. Very limited editing of the command line. |`Ex-mode`|

There are six ADDITIONAL modes. These are variants of the BASIC modes:

Operator-pending mode

This is like Normal mode, but after an operator command has started, and Vim is waiting for a `{motion}` to specify the text that the operator will work on.

Replace mode

Replace mode is a special case of Insert mode. You can do the same things as in Insert mode, but for each character you enter, one character of the existing text is deleted. See |`:h Replace-mode`|. If the `'showmode'` option is on `-- REPLACE --` is shown at the bottom of the window.

Virtual Replace mode

Virtual Replace mode is similar to Replace mode, but instead of file characters you are replacing screen real estate. See |`:h Virtual-Replace-mode`|. If the `'showmode'` option is on `-- VREPLACE --` is shown at the bottom of the window.

Insert Normal mode

Entered when CTRL-O given in Insert mode. This is like Normal mode, but after executing one command Vim returns to Insert mode. If the `'showmode'` option is on `-- (insert) --` is shown at the bottom of the window.

Insert Visual mode

Entered when starting a Visual selection from Insert mode, e.g., by using CTRL-O and then "v", "V" or CTRL-V. When the Visual selection ends, Vim returns to Insert mode. If the `'showmode'` option is on `-- (insert) VISUAL --` is shown at the bottom of the window.

Insert Select mode

Entered when starting Select mode from Insert mode. E.g., by dragging the mouse or <S-Right>. When the Select mode ends, Vim returns to Insert mode. If the `'showmode'` option is on `-- (insert) SELECT --` is shown at the bottom of the window.

Switching from mode to mode

If for any reason you do not know which mode you are in, you can always get back to Normal mode by typing <Esc> twice. This doesn't work for Ex mode though, use ":visual". You will know you are back in Normal mode when you see the screen flash or hear the bell after you type <Esc>. However, when pressing <Esc> after using CTRL-O in Insert mode you get a beep but you are still in Insert mode, type <Esc> again.

FROM mode \ TO mode	Normal	Visual	Select	Insert	Replace	Cmd-line	Ex
Normal		v V ^V	*4	*1	R gR	: / ? !	Q
Visual	*2		^G	c C	–	:	–
Select *5	^O ^G		*6	–	–	–	
Insert	<Esc>	–	–		<Insert>	–	–
Replace	<Esc>	–	–	<Insert>		–	–
Command-line	*3	–	–	:start	–		–
Ex	:vi	–	–	–	–	–	

– not possible

- *1 Go from Normal mode to Insert mode by giving the command ”i”, ”I”, ”a”, ”A”, ”o”, ”O”, ”c”, ”C”, ”s” or ”S”.
- *2 Go from Visual mode to Normal mode by giving a non-movement command, which causes the command to be executed, or by hitting <Esc> ”v”, ”V” or ”CTRL-V” (see |`:h v_v`|), which just stops Visual mode without side effects.
- *3 Go from Command-line mode to Normal mode by:
  - Hitting <CR> or <NL>, which causes the entered command to be executed.
  - Deleting the complete line (e.g., with CTRL-U) and giving a final <BS>.
  - Hitting CTRL-C or <Esc>, which quits the command-line without executing the command.

  In the last case <Esc> may be the character defined with the `'wildchar'` option, in which case it will start command-line completion. You can ignore that and type <Esc> again. {Vi: when hitting <Esc> the command-line is executed. This is unexpected for most people; therefore it was changed in Vim. But when the <Esc> is part of a mapping, the command-line is executed. If you want the Vi behaviour also when typing <Esc>, use ”`:cmap ^V<Esc> ^V^M`”}
- *4 Go from Normal to Select mode by:
  - use the mouse to select text while `'selectmode'` contains ”mouse”
  - use a non-printable command to move the cursor while keeping the Shift key pressed, and the `'selectmode'` option contains ”key”
  - use ”v”, ”V” or ”CTRL-V” while `'selectmode'` contains ”cmd”
  - use ”gh”, ”gH” or ”g CTRL-H” |`:h g_CTRL-H`|
- *5 Go from Select mode to Normal mode by using a non-printable command to move the cursor, without keeping the Shift key pressed.
- *6 Go from Select mode to Insert mode by typing a printable character. The selection is deleted and the character is inserted.

If the `'insertmode'` option is on, editing a file will start in Insert mode.

Additionally the command CTRL-\CTRL-N or <C-\><C-N> can be used to go to Normal mode from any other mode. This can be used to make sure Vim is in Normal mode, without causing a beep like <Esc> would. However, this does not work in Ex mode. When used after a command that takes an argument, such as |f| or |`:h m`|, the timeout set with `ttimeoutlen` applies.

The command CTRL-\CTRL-G or <C-\><C-G> can be used to go to Insert mode when `'insertmode'` is set. Otherwise it goes to Normal mode. This can be used to make sure Vim is in the mode indicated by `'insertmode'`, without knowing in what mode Vim currently is.

Q

Switch to ”Ex” mode. This is a bit like typing ”:” commands one after another, except:

- You don’t have to keep pressing ”:”.
- The screen doesn’t get updated after each command.
- There is no normal command-line editing.
- Mappings and abbreviations are not used.

In fact, you are editing the lines with the "standard" line-input editing commands (<Del> or <BS> to erase, CTRL-U to kill the whole line). Vim will enter this mode by default if it's invoked as "ex" on the command-line. Use the "`:vi`" command |`:h :visual`| to exit "Ex" mode. Note: In older versions of Vim "Q" formatted text, that is now done with |`:h gq`|. But if you use the |`:h vimrc_example.vim`| script "Q" works like "gq".

gQ

Switch to "Ex" mode like with "Q", but really behave like typing ":" commands after another. All command line editing, completion etc. is available. Use the "`:vi`" command |`:h :visual`| to exit "Ex" mode. {not in Vi}

The window contents

In Normal mode and Insert/Replace mode the screen window will show the current contents of the buffer: What You See Is What You Get. There are two exceptions:

- When the `'cpoptions'` option contains `$`, and the change is within one line, the text is not directly deleted, but a `$ is` put at the last deleted character.
- When inserting text in one window, other windows on the same text are not updated until the insert is finished.

{Vi: The screen is not always updated on slow terminals}

Lines longer than the window width will wrap, unless the `'wrap'` option is off (see below). The `'linebreak'` option can be set to wrap at a blank character.

If the window has room after the last line of the buffer, Vim will show `~` in the first column of the last lines in the window, like this:

```
+-----------------------+
|some line              |
|last line              |
|~                      |
|~                      |
+-----------------------+
```

Thus the `~` lines indicate that the end of the buffer was reached.

If the last line in a window doesn't fit, Vim will indicate this with a `@` in the first column of the last lines in the window, like this:

```
+-----------------------+
|first line             |
|second line            |
|@                      |
|@                      |
+-----------------------+
```

Thus the '@' lines indicate that there is a line that doesn't fit in the window.

When the "lastline" flag is present in the `'display'` option, you will not see `@` characters at the left side of window. If the last line doesn't fit completely, only the part that fits is shown, and the last three characters of the last line are replaced with "@@@", like this:

```
+----------------------+
|first line            |
|second line           |
|a very long line that d|
|oesn't fit in the wi@@@|
+----------------------+
```

If there is a single line that is too long to fit in the window, this is a special situation. Vim will show only part of the line, around where the cursor is. There are no special characters shown, so that you can edit all parts of this line. {Vi: gives an "internal error" on lines that do not fit in the window}

The '@' occasion in the `'highlight'` option can be used to set special highlighting for the '@' and `~` characters. This makes it possible to distinguish them from real characters in the buffer.

The `'showbreak'` option contains the string to put in front of wrapped lines.

If the `'wrap'` option is off, long lines will not wrap. Only the part that fits on the screen is shown. If the cursor is moved to a part of the line that is not shown, the screen is scrolled horizontally. The advantage of this method is that columns are shown as they are and lines that cannot fit on the screen can be edited. The disadvantage is that you cannot see all the characters of a line at once. The `'sidescroll'` option can be set to the minimal number of columns to scroll. {Vi: has no `'wrap'` option}

All normal ASCII characters are displayed directly on the screen. The <Tab> is replaced with the number of spaces that it represents. Other non-printing characters are replaced with "^`{char}`", where `{char}` is the non-printing character with 64 added. Thus character 7 (bell) will be shown as "^G". Characters between 127 and 160 are replaced with "~{char}", where {char} is the character with 64 subtracted. These characters occupy more than one position on the screen. The cursor can only be positioned on the first one.

If you set the `'number'` option, all lines will be preceded with their number. Tip: If you don't like wrapping lines to mix with the line numbers, set the `'showbreak'` option to eight spaces:

```
:set showbreak=\ \ \ \ \ \ \ \
```

If you set the `'list'` option, <Tab> characters will not be shown as several spaces, but as "^I". A `$` will be placed at the end of the line, so you can find trailing blanks.

In Command-line mode only the command-line itself is shown correctly. The display of the buffer contents is updated as soon as you go back to Command mode.

The last line of the window is used for status and other messages. The status messages will only be used if an option is on:

status message	option	default	Unix default
current mode	`'showmode'`	on	on
command characters	`'showcmd'`	on	off
cursor position	`'ruler'`	off	off

The current mode is `-- INSERT --` or `-- REPLACE --`, see |`:h 'showmode'`|. The command characters are those that you typed but were not used yet. {Vi: does not show the characters you typed or the cursor position}

If you have a slow terminal you can switch off the status messages to speed up editing:

```
:set nosc noru nosm
```

If there is an error, an error message will be shown for at least one second (in reverse video). {Vi: error messages may be overwritten with other messages before you have a chance to read them}

Some commands show how many lines were affected. Above which threshold this happens can be controlled with the `'report'` option (default 2).

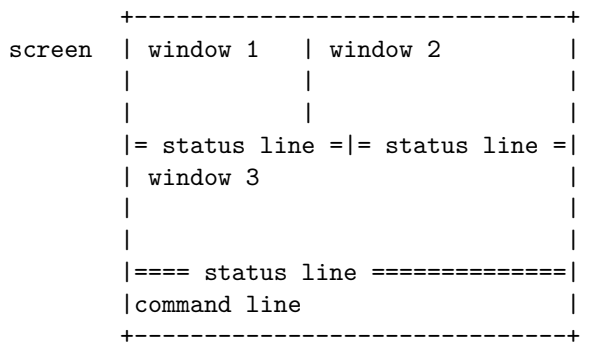
On the Amiga Vim will run in a CLI window. The name Vim and the full name of the current file name will be shown in the title bar. When the window is resized, Vim will automatically redraw the window. You may make the window as small as you like, but if it gets too small not a single line will fit in it. Make it at least 40 characters wide to be able to read most messages on the last line.

On most Unix systems, resizing the window is recognized and handled correctly by Vim. {Vi: not ok}

Definitions

screen	The whole area that Vim uses to work in. This can be a terminal emulator window. Also called "the Vim window".
window	A view on a buffer.

A screen contains one or more windows, separated by status lines and with the command line at the bottom.



The command line is also used for messages. It scrolls up the screen when there is not enough room in the command line.

A difference is made between four types of lines:

buffer lines	The lines in the buffer. This is the same as the lines as they are read from/written to a file. They can be thousands of characters long.
logical lines	The buffer lines with folding applied. Buffer lines in a closed fold are changed to a single logical line: "+– 99 lines folded". They can be thousands of characters long.
window lines	The lines displayed in a window: A range of logical lines with wrapping, line breaks, etc. applied. They can only be as long as the width of the window allows, longer lines are wrapped or truncated.
screen lines	The lines of the screen that Vim uses. Consists of the window lines of all windows, with status lines and the command line added. They can only be as long as the width of the screen allows. When the command line gets longer it wraps and lines are scrolled to make room.

buffer lines	logical lines	window lines	screen lines
1. one	1. one	1. +– folded	1. +– folded
2. two	2. +– folded	2. five	2. five
3. three	3. five	3. six	3. six
4. four	4. six	4. seven	4. seven
5. five	5. seven		5. === status line ===
6. six			6. aaa
7. seven			7. bbb
			8. ccc ccc c
1. aaa	1. aaa	1. aaa	9. cc
2. bbb	2. bbb	2. bbb	10. ddd
3. ccc ccc ccc	3. ccc ccc ccc	3. ccc ccc c	11. ~
4. ddd	4. ddd	4. cc	12. === status line ===
		5. ddd	13. (command line)
		6. ~	

01. About the manuals

This chapter introduces the manuals available with Vim. Read this to know the conditions under which the commands are explained.

Two manuals

The Vim documentation consists of two parts:

The User manual Task oriented explanations, from simple to complex. Reads from start to end like a book.

The Reference manual Precise description of how everything in Vim works.

The notation used in these manuals is explained here: |`notation`|

Note for the LaTeXedition : In the original Vim manual, link are presented inbetween | symbol. This notation is kept here. But, those links can lead to another part of this manual or to the Vim reference manual. In the first case, the links work like any hyperlink you can find in a pdf document. Regarding the links leading to the reference manual, they are prefixed in this "edition" with `:h`. Typing this command inside of Vim should lead to the correct part of the reference manual. Some links to the User Manual were represented by the number corresponding to the subsection they were linked to. They're represented here by the title of the subsection instead. There is normally here a subsubsection about Jumping Around which have been deleted from the pdf edition because most of the things it explained didn't apply to a pdf file. However they do apply when reading the reference manual inside of Vim, which most users do. You can read the subsubsection by typing `:h 01.1` in Vim.

Vim installed

Most of the manuals assume that Vim has been properly installed. If you didn't do that yet, or if Vim doesn't run properly (e.g., files can't be found or in the GUI the menus do not show up) first read the chapter on installation: |`Installing Vim`|.

The manuals often assume you are using Vim with Vi-compatibility switched off. For most commands this doesn't matter, but sometimes it is important, e.g., for multi-level undo. An easy way to make sure you are using a nice setup is to copy the example vimrc file. By doing this inside Vim you don't have to check out where it is located. How to do this depends on the system you are using:

Unix:

```
:!cp -i $VIMRUNTIME/vimrc_example.vim ~/.vimrc
```

MS-DOS, MS-Windows, OS/2:

```
:!copy $VIMRUNTIME/vimrc_example.vim $VIM/_vimrc
```

Amiga:

```
:!copy $VIMRUNTIME/vimrc_example.vim $VIM/.vimrc
```

If the file already exists you probably want to keep it.

If you start Vim now, the `'compatible'` option should be off. You can check it with this command:

```
:set compatible?
```

If it responds with "nocompatible" you are doing well. If the response is "compatible" you are in trouble. You will have to find out why the option is still set. Perhaps the file you wrote above is not found. Use this command to find out:

```
:scriptnames
```

If your file is not in the list, check its location and name. If it is in the list, there must be some other place where the `'compatible'` option is switched back on.

For more info see |vimrc| and |`:h compatible-default`|.

Note:
This manual is about using Vim in the normal way. There is an alternative called "evim" (easy Vim). This is still Vim, but used in a way that resembles a click-and-type editor like Notepad. It always stays in Insert mode, thus it feels very different. It is not explained in the user manual, since it should be mostly self explanatory. See |`:h evim-keys`| for details.

Using the Vim tutor

Instead of reading the text (boring!) you can use the vimtutor to learn your first Vim commands. This is a 30 minute tutorial that teaches the most basic Vim functionality hands-on.

On Unix, if Vim has been properly installed, you can start it from the shell:

```
vimtutor
```

On MS-Windows you can find it in the Program/Vim menu. Or execute `vimtutor.bat` in the $VIMRUNTIME directory.

This will make a copy of the tutor file, so that you can edit it without the risk of damaging the original. There are a few translated versions of the tutor. To find out if yours is available, use the two-letter language code. For French:

```
vimtutor fr
```

On Unix, if you prefer using the GUI version of Vim, use "gvimtutor" or "vimtutor -g" instead of "vimtutor".

For OpenVMS, if Vim has been properly installed, you can start vimtutor from a VMS prompt with:

```
@VIM:vimtutor
```

Optionally add the two-letter language code as above.

On other systems, you have to do a little work:

1. Copy the tutor file. You can do this with Vim (it knows where to find it):

   ```
   vim -u NONE -c 'e $VIMRUNTIME/tutor/tutor' -c 'w! TUTORCOPY' -c 'q'
   ```

   This will write the file "TUTORCOPY" in the current directory. To use a translated version of the tutor, append the two-letter language code to the filename. For French:

   ```
   vim -u NONE -c 'e $VIMRUNTIME/tutor/tutor.fr' -c 'w! TUTORCOPY' -c 'q'
   ```

2. Edit the copied file with Vim:

   ```
   vim -u NONE -c "set nocp" TUTORCOPY
   ```

   The extra arguments make sure Vim is started in a good mood.

3. Delete the copied file when you are finished with it:

```
del TUTORCOPY
```

Copyright

Parts of the user manual come from the book *Vi IMproved - Vim* by Steve Oualline (published by New Riders Publishing, ISBN: 0735710015). The Open Publication License applies to this book. Only selected parts are included and these have been modified (e.g., by removing the pictures, updating the text for Vim 6.0 and later, fixing mistakes). The omission of the |`frombook`| tag does not mean that the text does not come from the book.

Many thanks to Steve Oualline and New Riders for creating this book and publishing it under the OPL! It has been a great help while writing the user manual. Not only by providing literal text, but also by setting the tone and style.

If you make money through selling the manuals, you are strongly encouraged to donate part of the profit to help AIDS victims in Uganda. See |`:h iccf`|.

02. The first steps in Vim

This chapter provides just enough information to edit a file with Vim. Not well or fast, but you can edit. Take some time to practice with these commands, they form the base for what follows.

Running Vim for the First Time

To start Vim, enter this command:

```
gvim file.txt
```

In UNIX you can type this at any command prompt. If you are running Microsoft Windows, open an MS-DOS prompt window and enter the command. In either case, Vim starts editing a file called file.txt. Because this is a new file, you get a blank window. This is what your screen will look like:

```
+----------------------------------------+
|#                                       |
|~                                       |
|~                                       |
|~                                       |
|~                                       |
|"file.txt" [New file]                   |
+----------------------------------------+
                ('#" is the cursor position.)
```

The tilde (~) lines indicate lines not in the file. In other words, when Vim runs out of file to display, it displays tilde lines. At the bottom of the screen, a message line indicates the file is named file.txt and shows that you are creating a new file. The message information is temporary and other information overwrites it.

The vim command

The gvim command causes the editor to create a new window for editing. If you use this command:

```
vim file.txt
```

the editing occurs inside your command window. In other words, if you are running inside an xterm, the editor uses your xterm window. If you are using an MS-DOS command prompt window under Microsoft Windows, the editing occurs inside this window. The text in the window will look the same for both versions, but with gvim you have extra features, like a menu bar. More about that later.

Inserting text

The Vim editor is a modal editor. That means that the editor behaves differently, depending on which mode you are in. The two basic modes are called Normal mode and Insert mode. In Normal mode the characters you type are commands. In Insert mode the characters are inserted as text. Since you have just started Vim it will be in Normal mode. To start Insert mode you type the ”i” command (i for Insert). Then you can enter the text. It will be inserted into the file. Do not worry if you make mistakes; you can correct them later. To enter the following programmer’s limerick, this is what you type:

```
iA very intelligent turtle
Found programming UNIX a hurdle
```

After typing "turtle" you press the <Enter> key to start a new line. Finally you press the <Esc> key to stop Insert mode and go back to Normal mode. You now have two lines of text in your Vim window:

```
+----------------------------------------+
|A very intelligent turtle               |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

What is the mode?

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

```
:set showmode
```

You will notice that when typing the colon Vim moves the cursor to the last line of the window. That's where you type colon commands (commands that start with a colon). Finish this command by pressing the <Enter> key (all commands that start with a colon are finished this way). Now, if you type the "i" command Vim will display –INSERT– at the bottom of the window. This indicates you are in Insert mode.

```
+----------------------------------------+
|A very intelligent turtle               |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|-- INSERT --                            |
+----------------------------------------+
```

If you press <Esc> to go back to Normal mode the last line will be made blank.

Getting out of trouble

One of the problems for Vim novices is mode confusion, which is caused by forgetting which mode you are in or by accidentally typing a command that switches modes. To get back to Normal mode, no matter what mode you are in, press the <Esc> key. Sometimes you have to press it twice. If Vim beeps back at you, you already are in Normal mode.

Moving around

After you return to Normal mode, you can move around by using these keys:

h	left
j	down
k	up
l	right

At first, it may appear that these commands were chosen at random. After all, who ever heard of using l for right? But actually, there is a very good reason for these choices: Moving the cursor is the most common thing you do in an editor, and these keys are on the home row of your right hand. In other words, these commands are placed where you can type them the fastest (especially when you type with ten fingers).

Note: You can also move the cursor by using the arrow keys. If you do, however, you greatly slow down your editing because to press the arrow keys, you must move your hand from the text keys to the arrow keys. Considering that you might be doing it hundreds of times an hour, this can take a significant amount of time. Also, there are keyboards which do not have arrow keys, or which locate them in unusual places; therefore, knowing the use of the hjkl keys helps in those situations.

One way to remember these commands is that h is on the left, l is on the right and j points down. In a picture:

```
          k
        h   l
          j
```

The best way to learn these commands is by using them. Use the "i" command to insert some more lines of text. Then use the hjkl keys to move around and insert a word somewhere. Don't forget to press <Esc> to go back to Normal mode. The |`vimtutor`| is also a nice way to learn by doing.

For Japanese users, Hiroshi Iwatani suggested using this:

```
               Komsomolsk
                   ^
                   |
      Huan Ho  <--- --->  Los Angeles
   (Yellow river)  |
                   v
                 Java (the island, not the programming language)
```

Deleting characters

To delete a character, move the cursor over it and type "x". (This is a throwback to the old days of the typewriter, when you deleted things by typing xxxx over them.) Move the cursor to the beginning of the first line, for example, and type xxxxxxx (seven x's) to delete "A very ". The result should look like this:

```
+----------------------------------------+
|intelligent turtle                      |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

Now you can insert new text, for example by typing:

```
iA young <Esc>
```

This begins an insert (the i), inserts the words "A young", and then exits insert mode (the final <Esc>). The result:

```
+----------------------------------------+
|A young intelligent turtle              |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

Deleting a line

To delete a whole line use the "dd" command. The following line will then move up to fill the gap:

```
+----------------------------------------+
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

Deleting a line break

In Vim you can join two lines together, which means that the line break between them is deleted. The "J" command does this. Take these two lines:

```
A young intelligent
turtle
```

Move the cursor to the first line and press "J":

```
A young intelligent turtle
```

Undo and Redo

Suppose you delete too much. Well, you can type it in again, but an easier way exists. The "u" command undoes the last edit. Take a look at this in action: After using "dd" to delete the first line, "u" brings it back. Another one: Move the cursor to the A in the first line:

```
A young intelligent turtle
```

Now type xxxxxxx to delete "A young". The result is as follows:

```
intelligent turtle
```

Type "u" to undo the last delete. That delete removed the g, so the undo restores the character.

```
g intelligent turtle
```

The next u command restores the next-to-last character deleted:

```
ng intelligent turtle
```

The next u command gives you the u, and so on:

```
ung intelligent turtle
oung intelligent turtle
young intelligent turtle
young intelligent turtle
A young intelligent turtle
```

Note: If you type "u" twice, and the result is that you get the same text back, you have Vim configured to work Vi compatible. Look here to fix this: |`not-compatible`|. This text assumes you work "The Vim Way". You might prefer to use the good old Vi way, but you will have to watch out for small differences in the text then.

Redo

If you undo too many times, you can press CTRL-R (redo) to reverse the preceding command. In other words, it undoes the undo. To see this in action, press CTRL-R twice. The character A and the space after it disappear:

```
young intelligent turtle
```

There's a special version of the undo command, the "U" (undo line) command. The undo line command undoes all the changes made on the last line that was edited. Typing this command twice cancels the preceding "U".

```
A very intelligent turtle
  xxxx                      Delete very

A intelligent turtle
              xxxxxx        Delete turtle

A intelligent
                            Restore line with "U"
A very intelligent turtle
                            Undo "U" with "u"
A intelligent
```

The "U" command is a change by itself, which the "u" command undoes and CTRL-R redoes. This might be a bit confusing. Don't worry, with "u" and CTRL-R you can go to any of the situations you had. More about that in section |`Numbering changes`|.

Other editing commands

Vim has a large number of commands to change the text. See |`:h Q_in`| and below. Here are a few often used ones.

Appending

The "i" command inserts a character before the character under the cursor. That works fine; but what happens if you want to add stuff to the end of the line? For that you need to insert text after the cursor. This is done with the "a" (append) command. For example, to change the line

```
and that's not saying much for the turtle.
```

to

```
and that's not saying much for the turtle!!!
```

move the cursor over to the dot at the end of the line. Then type "x" to delete the period. The cursor is now positioned at the end of the line on the e in turtle. Now type

```
a!!!<Esc>
```

to append three exclamation points after the e in turtle:

```
and that's not saying much for the turtle!!!
```

Opening up a new line

The "o" command creates a new, empty line below the cursor and puts Vim in Insert mode. Then you can type the text for the new line. Suppose the cursor is somewhere in the first of these two lines:

```
A very intelligent turtle
Found programming UNIX a hurdle
```

If you now use the "o" command and type new text:

```
oThat liked using Vim<Esc>
```

The result is:

```
A very intelligent turtle
That liked using Vim
Found programming UNIX a hurdle
```

The "O" command (uppercase) opens a line above the cursor.

Using a count

Suppose you want to move up nine lines. You can type "kkkkkkkkk" or you can enter the command "9k". In fact, you can precede many commands with a number. Earlier in this chapter, for instance, you added three exclamation points to the end of a line by typing "a!!!<Esc>". Another way to do this is to use the command "3a!<Esc>". The count of 3 tells the command that follows to triple its effect. Similarly, to delete three characters, use the command "3x". The count always comes before the command it applies to.

Getting out

To exit, use the "ZZ" command. This command writes the file and exits.

Note:
Unlike many other editors, Vim does not automatically make a backup file. If you type "ZZ", your changes are committed and there's no turning back. You can configure the Vim editor to produce backup files, see |Backup files|.

Discarding changes

Sometimes you will make a sequence of changes and suddenly realize you were better off before you started. Not to worry; Vim has a quit-and-throw-things-away command. It is:

```
:q!
```

Don't forget to press <Enter> to finish the command.

For those of you interested in the details, the three parts of this command are the colon (:), which enters Command-line mode; the q command, which tells the editor to quit; and the override command modifier (!). The override command modifier is needed because Vim is reluctant to throw away changes. If you were to just type ":q", Vim would display an error message and refuse to exit:

```
E37: No write since last change (use ! to override)
```

By specifying the override, you are in effect telling Vim, "I know that what I'm doing looks stupid, but I'm a big boy and really want to do this."

If you want to continue editing with Vim: The "`:e!`" command reloads the original version of the file.

Finding help

Everything you always wanted to know can be found in the Vim help files. Don't be afraid to ask! To get generic help use this command:

```
:help
```

You could also use the first function key <F1>. If your keyboard has a <Help> key it might work as well. If you don't supply a subject, "`:help`" displays the general help window. The creators of Vim did something very clever (or very lazy) with the help system: They made the help window a normal editing window. You can use all the normal Vim commands to move through the help information. Therefore h, j, k, and l move left, down, up and right. To get out of the help window, use the same command you use to get out of the editor: "`ZZ`". This will only close the help window, not exit Vim.

As you read the help text, you will notice some text enclosed in vertical bars (for example, |`:h help`|). This indicates a hyperlink. If you position the cursor anywhere between the bars and press CTRL-] (jump to tag), the help system takes you to the indicated subject. (For reasons not discussed here, the Vim terminology for a hyperlink is tag. So CTRL-] jumps to the location of the tag given by the word under the cursor.) After a few jumps, you might want to go back. CTRL-T (pop tag) takes you back to the preceding position. CTRL-O (jump to older position) also works nicely here. At the top of the help screen, there is the notation *help.txt*. This name between "*" characters is used by the help system to define a tag (hyperlink destination). See |`Using tags`| for details about using tags.

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
```

The Vim editor has many different modes. By default, the help system displays the normal-mode commands. For example, the following command displays help for the normal-mode CTRL-H command:

```
:help CTRL-H
```

To identify other modes, use a mode prefix. If you want the help for the insert-mode version of a command, use "`i_`". For CTRL-H this gives you the following command:

```
:help i_CTRL-H
```

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
```

The Vim editor has a number of options that enable you to configure and customize the editor. If you want help for an option, you need to enclose it in single quotation marks. To find out what the `'number'` option does, for example, use the following command:

```
:help 'number'
```

The table with all mode prefixes can be found here: |`:h help-context`|.

Special keys are enclosed in angle brackets. To find help on the up-arrow key in Insert mode, for instance, use this command:

```
:help i_<Up>
```

If you see an error message that you don't understand, for example:

```
E37: No write since last change (use ! to override)
```

You can use the error ID at the start to find help about it:

```
:help E37
```

Summary

`:help`	Gives you very general help. Scroll down to see a list of all helpfiles, including those added locally (i.e. not distributed with Vim).
`:help user-toc.txt`	Table of contents of the User Manual.
`:help :subject`	Ex-command "subject", for instance the following:
`:help :help`	Help on getting help.
`:help abc`	normal-mode command "abc".
`:help CTRL-B`	Control key <C-B> in Normal mode.
`:help i_abc` `:help i_CTRL-B`	The same in Insert mode.
`:help v_abc` `:help v_CTRL-B`	The same in Visual mode.
`:help c_abc` `:help c_CTRL-B`	The same in Command-line mode.
`:help 'subject'`	Option `'subject'`.
`:help subject()`	Function "subject".
`:help -subject`	Command-line option "-subject".
`:help +subject`	Compile-time feature "+subject".
`:help EventName`	Autocommand event "EventName".
`:help digraphs.txt`	The top of the helpfile "digraph.txt". Similarly for any other helpfile.
`:help pattern<Tab>`	Find a help tag starting with "pattern". Repeat <Tab> for others.
`:help pattern<Ctrl-D>`	See all possible help tag matches "pattern" at once.
`:helpgrep pattern`	Search the whole text of all help files for pattern "pattern". Jumps to the first match. Jump to other matches with: • `:cn` next match • `:cprev` `:cN` previous match • `:cfirst` `:clast` first or last match • `:copen` `:cclose` open/close the quickfix window; press <Enter> to jump to the item under the cursor

03. Moving around

Before you can insert or delete text the cursor has to be moved to the right place. Vim has a large number of commands to position the cursor. This chapter shows you how to use the most important ones. You can find a list of these commands below |`:h Q_lr`|.

Word movement

To move the cursor forward one word, use the "w" command. Like most Vim commands, you can use a numeric prefix to move past multiple words. For example, "`3w`" moves three words.

This figure shows how it works:

```
This is a line with example text
  --->-->->----------------->
   w  w  w    3w
```

Notice that "`w`" moves to the start of the next word if it already is at the start of a word.

The "`b`" command moves backward to the start of the previous word:

```
This is a line with example text
<----<--<-<---------<---
   b   b b    2b      b
```

There is also the "`e`" command that moves to the next end of a word and "`ge`", which moves to the previous end of a word:

```
This is a line with example text
   <-   <--- ----->   ---->
   ge    ge     e       e
```

If you are at the last word of a line, the "`w`" command will take you to the first word in the next line. Thus you can use this to move through a paragraph, much faster than using "l". "`b`" does the same in the other direction.

A word ends at a non-word character, such as a ".", "-" or ")". To change what Vim considers to be a word, see the `'iskeyword'` option. It is also possible to move by white-space separated WORDs. This is not a word in the normal sense, that's why the uppercase is used. The commands for moving by WORDs are also uppercase, as this figure shows:

```
        ge      b          w                            e
        <-     <-         --->                         --->
This is-a line, with special/separated/words (and some more).
   <----- <-----         -------------------->         ----->
     gE      B                    W                     E
```

With this mix of lowercase and uppercase commands, you can quickly move forward and backward through a paragraph.

Moving to the start or end of a line

The `"$"` command moves the cursor to the end of a line. If your keyboard has an <End> key it will do the same thing.

The "^" command moves to the first non-blank character of the line. The "0" command (zero) moves to the very first character of the line. The <Home> key does the same thing. In a picture:

```
             ^
       <------------
.....This is a line with example text
<-----------------   --------------->
    0                      $
```

(the "....." indicates blanks here)

The "$" command takes a count, like most movement commands. But moving to the end of the line several times doesn't make sense. Therefore it causes the editor to move to the end of another line. For example, "1$" moves you to the end of the first line (the one you're on), "2$" to the end of the next line, and so on. The "0" command doesn't take a count argument, because the "0" would be part of the count. Unexpectedly, using a count with "^" doesn't have any effect.

Moving to a character

One of the most useful movement commands is the single-character search command. The command "fx" searches forward in the line for the single character x. Hint: "f" stands for "Find". For example, you are at the beginning of the following line. Suppose you want to go to the h of human. Just execute the command "fh" and the cursor will be positioned over the h:

```
To err is human.  To really foul up you need a computer.
---------->--------------->
    fh       fy
```

This also shows that the command "fy" moves to the end of the word really. You can specify a count; therefore, you can go to the "l" of "foul" with "3fl":

```
To err is human.  To really foul up you need a computer.
      --------------------->
           3fl
```

The "F" command searches to the left:

```
To err is human.  To really foul up you need a computer.
          <---------------------
               Fh
```

The "tx" command works like the "fx" command, except it stops one character before the searched character. Hint: "t" stands for "To". The backward version of this command is "Tx".

```
To err is human.  To really foul up you need a computer.
           <------------  ------------->
            Th              tn
```

These four commands can be repeated with ";". "," repeats in the other direction. The cursor is never moved to another line. Not even when the sentence continues.

Sometimes you will start a search, only to realize that you have typed the wrong command. You type "f" to search backward, for example, only to realize that you really meant "F". To abort a search, press <Esc>. So "f<Esc>" is an aborted forward search and doesn't do anything. Note: <Esc> cancels most operations, not just searches.

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 "(".

```
              %
            <----->
if (a == (b * c) / d)
   <----------------->
        %
```

This also works for [ ] and {} pairs. (This can be defined with the 'matchpairs' option.)

When the cursor is not on a useful character, "%" will search forward to find one. Thus if the cursor is at the start of the line of the previous example, "%" will search forward and find the first "(". Then it moves to its match:

```
if (a == (b * c) / d)
---+---------------->
       %
```

Moving to a specific line

If you are a C or C++ programmer, you are familiar with error messages such as the following:

```
prog.c:33: j   undeclared (first use in this function)
```

This tells you that you might want to fix something on line 33. So how do you find line 33? One way is to do "9999k" to go to the top of the file and "32j" to go down thirty two lines. It is not a good way, but it works. A much better way of doing things is to use the "G" command. With a count, this command positions you at the given line number. For example, "33G" puts you on line 33. (For a better way of going through a compiler's error list, see |Installing Vim|, for information on the :make command.) With no argument, "G" positions you at the end of the file. A quick way to go to the start of a file use "gg". "1G" will do the same, but is a tiny bit more typing.

```
    |  first line of a file   ^
    |  text text text text    |
    |  text text text text    |  gg
7G  |  text text text text    |
    |  text text text text
    |  text text text text
    V  text text text text    |
       text text text text    |  G
       text text text text    |
       last line of a file    V
```

Another way to move to a line is using the "%" command with a count. For example "50%" moves you to halfway the file. "90%" goes to near the end.

The previous assumes that you want to move to a line in the file, no matter if it's currently visible or not. What if you want to move to one of the lines you can see? This figure shows the three commands you can use:

```
          +---------------------------+
H -->     | text sample text          |
          | sample text               |
          | text sample text          |
          | sample text               |
M -->     | text sample text          |
          | sample text               |
          | text sample text          |
          | sample text               |
L -->     | text sample text          |
          +---------------------------+
```

Hints: ”H” stands for Home, ”M” for Middle and ”L” for Last.

Telling where you are

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

1. Use the CTRL-G command. You get a message like this (assuming the `'ruler'` option is off):

   ```
   "usr_03.txt" line 233 of 650 --35%-- col 45-52
   ```

   This shows the name of the file you are editing, the line number where the cursor is, the total number of lines, the percentage of the way through the file and the column of the cursor. 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.

2. Set the `'number'` option. This will display a line number in front of every line:

   ```
   :set number
   ```

   To switch this off again:

   ```
   :set nonumber
   ```

   Since `'number'` is a boolean option, prepending ”no” to its name has the effect of switching it off. A boolean option has only these two values, it is either on or off. Vim has many options. Besides the boolean ones there are options with a numerical value and string options. You will see examples of this where they are used.

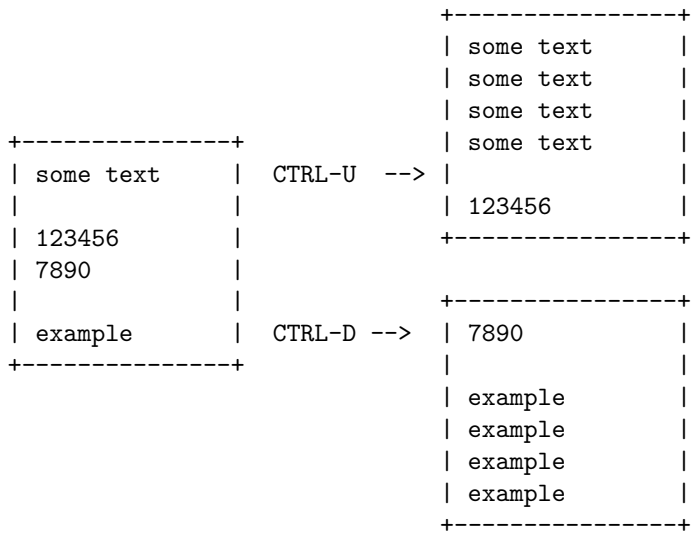
3. Set the `'ruler'` option. This will display the cursor position in the lower right corner of the Vim window:

   ```
   :set ruler
   ```

Using the `'ruler'` option has the advantage that it doesn’t take much room, thus there is more space for your text.

Scrolling around

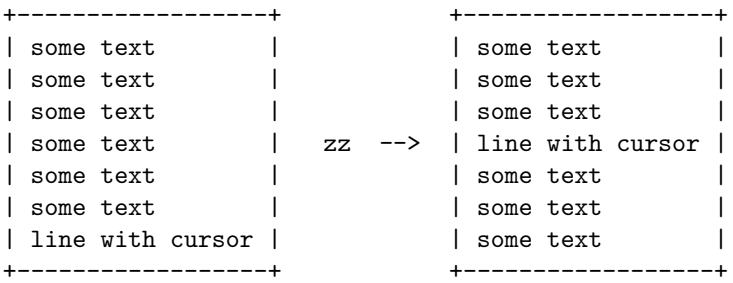
The CTRL-U command scrolls down half a screen of text. Think of looking through a viewing window at the text and moving this window up by half the height of the window. Thus the window moves up over the text, which is backward in the file. Don’t worry if you have a little trouble remembering which end is up. Most users have the same problem. The CTRL-D command moves the viewing window down half a screen in the file, thus scrolls the text up half a screen.



To scroll one line at a time use CTRL-E (scroll up) and CTRL-Y (scroll down). Think of CTRL-E to give you one line Extra. (If you use MS-Windows compatible key mappings CTRL-Y will redo a change instead of scroll.)

To scroll forward by a whole screen (except for two lines) use CTRL-F. The other way is backward, CTRL-B is the command to use. Fortunately CTRL-F is Forward and CTRL-B is Backward, that's easy to remember.

A common issue is that after moving down many lines with "j" your cursor is at the bottom of the screen. You would like to see the context of the line with the cursor. That's done with the "zz" command.



The "`zt`" command puts the cursor line at the top, "`zb`" at the bottom. There are a few more scrolling commands, see |`:h Q_sc`|. To always keep a few lines of context around the cursor, use the `'scrolloff'` option.

Simple searches

To search for a string, use the "`/string`" command. To find the word include, for example, use the command:

```
/include
```

You will notice that when you type the "`/`" the cursor jumps to the last line of the Vim window, like with colon commands. That is where you type the word. You can press the backspace key (backarrow or <BS>) to make corrections. Use the <Left> and <Right> cursor keys when necessary. Pressing <Enter> executes the command.

Note: The characters .*[ ]^%/\?~$ have special meanings. If you want to use them in a search you must put a \in front of them. See below.

To find the next occurrence of the same string use the "n" command. Use this to find the first #include after the cursor:

```
/#include
```

And then type "n" several times. You will move to each #include in the text. You can also use a count if you know which match you want. Thus "3n" finds the third match. Using a count with "/" doesn't work.

The "?" command works like "/" but searches backwards:

```
?word
```

The "N" command repeats the last search the opposite direction. Thus using "N" after a "/" command search backwards, using "N" after "?" searches forward.

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 ttexhe 'ignorecase' option:

```
:set ignorecase
```

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

```
:set noignorecase
```

History

Suppose you do three searches:

```
/one
/two
/three
```

Now let's start searching by typing a simple "/" without pressing <Enter>. If you press <Up> (the cursor key), Vim puts "/three" on the command line. Pressing <Enter> at this point searches for three. If you do not press <Enter>, but press <Up> instead, Vim changes the prompt to "/two". Another press of <Up> moves you to "/one". You can also use the <Down> cursor key to move through the history of search commands in the other direction.

If you know what a previously used pattern starts with, and you want to use it again, type that character before pressing <Up>. With the previous example, you can type "/o<Up>" and Vim will put "/one" on the command line.

The commands starting with ":" also have a history. That allows you to recall a previous command and execute it again. These two histories are separate.

Searching for a word in the text

Suppose you see the word "TheLongFunctionName" in the text and you want to find the next occurrence of it. You could type "/TheLongFunctionName", but that's a lot of typing. And when you make a mistake Vim won't find it. There is an easier way: 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.

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).

Highlighting matches

While editing a program you see a variable called "`nr`". You want to check where it's used. You could move the cursor to "`nr`" and use the "`*`" command and press "`n`" to go along all the matches. There is another way. Type this command:

```
:set hlsearch
```

If you now search for "`nr`", Vim will highlight all matches. That is a very good way to see where the variable is used, without the need to type commands. To switch this off:

```
:set nohlsearch
```

Then you need to switch it on again if you want to use it for the next search command. If you only want to remove the highlighting, use this command:

```
: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.

Tuning searches

There are a few options that change how searching works. These are the essential ones:

```
: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.

Intermezzo

If you like one of the options mentioned before, and set it each time you use Vim, you can put the command in your Vim startup file. 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:

```
ZZ
```

If you now start Vim again, the `'hlsearch'` option will already be set.

Simple search patterns

The Vim editor uses regular expressions to specify what to search for. Regular expressions are an extremely powerful and compact way to specify a search pattern. Unfortunately, this power comes at a price, because regular expressions are a bit tricky to specify. In this section we mention only a few essential ones. More about search patterns and commands in chapter 27 |`Search commands and patterns`|. You can find the full explanation here: |`:h pattern`|.

Beginning and end of a line

The `^` character matches the beginning of a line. On an English-US keyboard you find it above the 6. The pattern "`include`" matches the word include anywhere on the line. But the pattern "`^include`" matches the word include only if it is at the beginning of a line. The $ character matches the end of a line. Therefore, "`was$`" matches the word was only if it is at the end of a line.

Let's mark the places where "the" matches in this example line with "x"s:

```
the solder holding one of the chips melted and the
xxx                      xxx                  xxx
```

Using "`/the$`" we find this match:

```
the solder holding one of the chips melted and the
                                                xxx
```

And with "`/^the`" we find this one:

```
the solder holding one of the chips melted and the
xxx
```

You can try searching with "`/^the$`", it will only match a single line consisting of "the". White space does matter here, thus if a line contains a space after the word, like "the ", the pattern will not match.

Matching any single character

The . (dot) character matches any existing character. For example, the pattern "`c.m`" matches a string whose first character is a c, whose second character is anything, and whose the third character is m. Example:

```
We use a computer that became the cummin winter.
         xxx             xxx      xxx
```

Matching special characters

If you really want to match a dot, you must avoid its special meaning by putting a backslash before it. If you search for "`ter.`", you will find these matches:

```
We use a computer that became the cummin winter.
              xxxx                          xxxx
```

Searching for "`ter\.`" only finds the second match.

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 mark. To go back where you came from, use this command:

```
``
```

This `` ` `` is a backtick or open single-quote character. If you use the same command a second time you will jump back again. That's because the `` ` `` command is a jump itself, and the position from before this jump is remembered.

Generally, every time you do a command that can move the cursor further than within the same line, this is called a jump. This includes the search commands "`/`" and "`n`" (it doesn't matter how far away the match is). But not the character searches with "fx" and "tx" or the word movements "w" and "e". Also, "j" and "k" are not considered to be a jump. Even when you use a count to make them move the cursor quite a long way away.

The `` `` `` command jumps back and forth, between two points. 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). Consider this sequence of commands:

```
33G
/^The
CTRL-O
```

You first jump to line 33, then search for a line that starts with "The". Then with CTRL-O you jump back to line 33. Another CTRL-O takes you back to where you started. If you now use CTRL-I you jump to line 33 again. And to the match for "The" with another CTRL-I.

```
       | example text   ^             |
  33G  | example text   |  CTRL-O     | CTRL-I
       | example text   |             |
       V line 33 text   ^             V
       | example text   |             |
/^The  | example text   |  CTRL-O     | CTRL-I
       V There you are  |             V
         example text
```

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 ">".

Named marks

Vim enables you to place your own marks in the text. The command "`ma`" marks the place under the cursor as mark a. You can place 26 marks (a through z) in your text. You can't see them, it's just a position that Vim remembers. To go to a mark, use the command `` `{mark} ``, where `{mark}` is the mark letter. Thus to move to the a mark:

```
`a
```

The command `'mark` (single quotation mark, or apostrophe) moves you to the beginning of the line containing the mark. This differs from the `` `mark `` command, which moves you to marked column.

The marks can be very useful when working on two related parts in a file. Suppose you have some text near the start of the file you need to look at, while working on some text near the end of the file. Move to the text at the start and place the s (start) mark there:

```
ms
```

Then move to the text you want to work on and put the e (end) mark there:

```
me
```

Now you can move around, and when you want to look at the start of the file, you use this to jump there:

```
's
```

Then you can use `''` to jump back to where you were, or `'e` to jump to the text you were working on at the end. There is nothing special about using s for start and e for end, they are just easy to remember.

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

04. Making small changes

This chapter shows you several ways of making corrections and moving text around. It teaches you the three basic ways to change text: operator-motion, Visual mode and text objects.

Operators and motions

In chapter 2 you learned the "`x`" command to delete a single character. And using a count: "`4x`" deletes four characters.

The "`dw`" command deletes a word. You may recognize the "`w`" command as the move word command. In fact, the "`d`" command may be followed by any motion command, and it deletes from the current location to the place where the cursor winds up.

The "`4w`" command, for example, moves the cursor over four words. The "`d4w`" command deletes four words.

```
To err is human. To really foul up you need a computer.
                 ------------------>
                       d4w

To err is human. you need a computer.
```

Vim only deletes up to the position where the motion takes the cursor. That's because Vim knows that you probably don't want to delete the first character of a word. If you use the "e" command to move to the end of a word, Vim guesses that you do want to include that last character:

```
To err is human. you need a computer.
                --------->
                   d2e

To err is human. a computer.
```

Whether the character under the cursor is included depends on the command you used to move to that character. The reference manual calls this "exclusive" when the character isn't included and "inclusive" when it is.

The "`$`" command moves to the end of a line. The "`d$`" command deletes from the cursor to the end of the line. This is an inclusive motion, thus the last character of the line is included in the delete operation:

```
To err is human. a computer.
                ------------>
                    d$

To err is human
```

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.

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.

```
To err is human
   ------->
     c2wbe<Esc>

To be human
```

This "`c2wbe<Esc>`" contains these bits:

c	the change operator
2w	move two words (they are deleted and Insert mode started)
be	insert this text
<Esc>	back to Normal mode

If you have paid attention, you will have noticed something strange: The space before "human" isn't deleted. There is a saying that for every problem there is an answer that is simple, clear, and wrong. That is the case with the example used here for the "`cw`" command. The c operator works just like the d operator, with one exception: "`cw`". It actually works like "`ce`", change to end of word. Thus the space after the word isn't included. This is an exception that dates back to the old Vi. Since many people are used to it now, the inconsistency has remained in Vim.

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.

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)

Where to put the count

The commands "`3dw`" and "`d3w`" 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.

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>

```
there is somerhing grong here
rT           rt    rw

There is something wrong here
```

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

```
There is something wrong here
                  5rx

There is something xxxxx here
```

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.

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.

```
                    To <B>generate</B> a table of <B>contents
f<   find first <   --->
df>  delete to >       -->
f<   find next <         --------->
.    repeat df>                 --->
f<   find next <                   ------------->
.    repeat df>                                 -->
```

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.

Visual mode

To delete simple items the operator-motion changes work quite well. But often it's not so easy to decide which command will move over the text you want to change. Then you can use Visual mode.

You start Visual mode by pressing "v". You move the cursor over the text you want to work on. While you do this, the text is highlighted. Finally type the operator command. For example, to delete from halfway one word to halfway another word:

```
This is an examination sample of visual mode
                ---------->
                  velllld

This is an example of visual mode
```

When doing this you don't really have to count how many times you have to press "l" to end up in the right position. You can immediately see what text will be deleted when you press "d".

If at any time you decide you don't want to do anything with the highlighted text, just press <Esc> and Visual mode will stop without doing anything.

Selecting lines

If you want to work on whole lines, use "`V`" to start Visual mode. You will see right away that the whole line is highlighted, without moving around. When you move left or right nothing changes. When you move up or down the selection is extended whole lines at a time. For example, select three lines with "`Vjj`":

```
                   +-------------------------+
                   | text more text          |
                >> | more text more text     | |
selected lines  >> | text text text          | | Vjj
                >> | text more               | V
                   | more text more          |
                   +-------------------------+
```

Selecting blocks

If you want to work on a rectangular block of characters, use CTRL-V to start Visual mode. This is very useful when working on tables.

```
name        Q1  Q2  Q3
pierre      123 455 234
john        0   90  39
steve       392 63  334
```

To delete the middle "Q2" column, move the cursor to the "Q" of "Q2". Press CTRL-V to start blockwise Visual mode. Now move the cursor three lines down with "`3j`" and to the next word with "`w`". You can see the first character of the last column is included. To exclude it, use "`h`". Now press "`d`" and the middle column is gone.

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.

Note that "`o`" and "`O`" in Visual mode work very differently from Normal mode, where they open a new line below or above the cursor.

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).

Take a look at how this works. First you will delete an entire line, by putting the cursor on the line you want to delete and typing "`dd`". Now you move the cursor to where you want to put the line and use the "`p`" (put) command. The line is inserted on the line below the cursor.

```
a line         a line        a line
line 2    dd   line 3    p   line 3
line 3                       line 2
```

Because you deleted an entire line, the "`p`" command placed the text line below the cursor. If you delete part of a line (a word, for instance), the "`p`" command puts it just after the cursor.

```
Some more boring try text to out commands.
                 ---->
                  dw

Some more boring text to out commands.
                 ------->
                   welp

Some more boring text to try out commands.
```

More on putting

The "`P`" command puts text like "`p`", but before the cursor. When you deleted a whole line with "`dd`", "`P`" will put it back above the cursor. When you deleted a word with "`dw`", "`P`" will put it back just before the cursor.

You can repeat putting as many times as you like. The same text will be used.

You can use a count with "`p`" and "`P`". The text will be repeated as many times as specified with the count. Thus "`dd`" and then "`3p`" puts three copies of the same deleted line.

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 "teh" 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.

```
teh     th     the
 x       p
```

Copying text

To copy text from one place to another, you could delete it, use "`u`" to undo the deletion and then "`p`" to put it somewhere else. There is an easier way: yanking. The "`y`" operator copies text into a register. Then a "`p`" command can be used to put it.

Yanking is just a Vim name for copying. The "c" letter was already used for the change operator, and "y" was still available. Calling this operator "yank" made it easier to remember to use the "y" key.

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

```
let sqr = LongVariable *
         -------------->
               y2w

let sqr = LongVariable *
                       p

let sqr = LongVariable * LongVariable
```

Notice that "yw" includes the white space after a word. If you don't want this, use "ye".

The "yy" command yanks a whole line, just like "dd" deletes a whole line. Unexpectedly, while "D" deletes from the cursor to the end of the line, "Y" works like "yy", it yanks the whole line. Watch out for this inconsistency! Use "y$" to yank to the end of the line.

```
a text line   yy   a text line        a text line
line 2             line 2         p   line 2
last line          last line          a text line
                                      last line
```

Using the clipboard

If you are using the GUI version of Vim (gvim), you can find the "Copy" item in the "Edit" menu. First select some text with Visual mode, then use the Edit/Copy menu. The selected text is now copied to the clipboard. You can paste the text in other programs. In Vim itself too.

If you have copied text to the clipboard in another application, you can paste it in Vim with the Edit/Paste menu. This works in Normal mode and Insert mode. In Visual mode the selected text is replaced with the pasted text.

The "Cut" menu item deletes the text before it's put on the clipboard. The "Copy", "Cut" and "Paste" items are also available in the popup menu (only when there is a popup menu, of course). If your Vim has a toolbar, you can also find these items there.

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
```

This only works on versions of Vim that include clipboard support. More about the clipboard in section 09.3 and here: |:h clipboard|.

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`".

```
this is some example text.
           daw

this is some text.
```

The "`d`" of "`daw`" is the delete operator. "`aw`" is a text object. 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).

Using text objects is the third way to make changes in Vim. We already had operator-motion and Visual mode. Now we add operator-text object.

It is very similar to operator-motion, but instead of operating on the text between the cursor position before and after a movement command, the text object is used as a whole. It doesn't matter where in the object the cursor was.

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

```
Hello there.  This
is an example.  Just
some text.
```

Move to the start of the second line, on "is an". Now use "`cis`":

```
Hello there.    Just
some text.
```

The cursor is in between the blanks in the first line. Now you type the new sentence "Another line.":

```
Hello there.  Another line.  Just
some 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`".

You can also use text objects in Visual mode. It will include the text object in the Visual selection. Visual mode continues, thus you can do this several times. For example, start Visual mode with "`v`" and select a sentence with "`as`". Now you can repeat "`as`" to include more sentences. Finally you use an operator to do something with the selected sentences.

You can find a long list of text objects here: |`:h text-objects`|.

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>.

In this example you start Replace mode on the first "t" of "text":

```
This is text.
        Rinteresting.<Esc>

This is interesting.
```

You may have noticed that this command replaced 5 characters in the line with twelve others. The "`R`" command automatically extends the line if it runs out of characters to replace. It will not continue on the next line.

You can switch between Insert mode and Replace mode with the <Insert> key.

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.

Conclusion

The operators, movement commands and text objects give you the possibility to make lots of combinations. Now that you know how it works, you can use N operators with M movement commands to make N * M commands!

You can find a list of operators here: |`:h operator`|

For example, there are many other ways to delete pieces of text. Here are a few often used ones:

x	delete character under the cursor (short for "`dl`")
X	delete character before the cursor (short for "`dh`")
D	delete from cursor to end of line (short for "`d$`")
dw	delete from cursor to next start of word
db	delete from cursor to previous start of word
diw	delete word under the cursor (excluding white space)
daw	delete word under the cursor (including white space)
dG	delete until the end of the file
dgg	delete until the start of the file

If you use "`c`" instead of "`d`" they become change commands. And with "`y`" you yank the text. And so forth.

There are a few often used commands to make changes that didn't fit somewhere else:

- ~ change case of the character under the cursor, and move the cursor to the next character. This is not an operator (unless `'tildeop'` is set), thus you can't use it with a motion command. It does work in Visual mode and changes case for all the selected text then.
- `I` Start Insert mode after moving the cursor to the first non-blank in the line.
- `A` Start Insert mode after moving the cursor to the end of the line.

05. Set your settings

Vim can be tuned to work like you want it to. This chapter shows you how to make Vim start with options set to different values. Add plugins to extend Vim's capabilities. Or define your own macros.

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
```

If you don't have a vimrc file yet, see |`:h vimrc`| to find out where you can create a vimrc file. Also, the "`:version`" command mentions the name of the "user vimrc file" Vim looks for.

For Unix and Macintosh this file is always used and is recommended:

```
~/.vimrc
```

For MS-DOS and MS-Windows you can use one of these:

```
$HOME/_vimrc
$VIM/_vimrc
```

The vimrc file can contain all the commands that you type after a colon. The most simple ones are for setting options. For example, if you want Vim to always start with the 'incsearch' option on, add this line you your vimrc file:

```
set incsearch
```

For this new line to take effect you need to exit Vim and start it again. Later you will learn how to do this without exiting Vim.

This chapter only explains the most basic items. For more information on how to write a Vim script file: |`Write a Vim script`|.

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. See |Backup files|

```
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.

```
    +-----------------------------------------------+
    |text in the Vim window                         |
    |~                                              |
    |~                                              |
    |-- VISUAL --                  2f      43,8   17% |
    +-----------------------------------------------+
     ^^^^^^^^^^                  ^^^^^^^ ^^^^^^^^^^
      'showmode'                 'showcmd' 'ruler'
```

```
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. More about that in |`Write a Vim script`|.

```
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. See |`:h filetypes`|.
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, see |`write-filetype-plugin`|.
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 |`:h :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.

```
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 |`:h line-continuation`|. This only works in a Vim script file, not when typing commands at the command-line.

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. Watch out for this difference when reading the manual!

Let's break this down:

`<F5>`	The F5 function key. This is the trigger key that causes the command to be executed as the key is pressed.
`i{<Esc>`	Insert the { character. The <Esc> key ends Insert mode.
`e`	Move to the end of the word.
`a}<Esc>`	Append the } to the word.

After you execute the "`:map`" command, all you have to do to put { } around a word is to put the cursor on the first character and press F5.

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>
```

You need to type the `\` and the `p` quickly after another, so that Vim knows they belong together.

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

Adding a package

(This is missing. Was it intentional?)

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 |`:h +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

The global plugins will be discussed first, then the filetype ones |`add-filetype-plugin`|.

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 |`:h standard-plugin-list`|. Also see |`:h load-plugins`|.

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.

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 |`:h vimball`|. Some plugins can be updated automatically, see |`:h getscript`|.

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	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
```

That's all! Now you can use the commands defined in this plugin to justify text.

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.

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
```

That's all! See |`vimrc-filetype`|.

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.

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`.

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.

On MS-DOS you cannot use long filenames. You would run into trouble if you add a second plugin and the filetype has more than six characters. You can use an extra directory to get around this:

```
mkdir $VIM/vimfiles/ftplugin/fortran
copy thefile $VIM/vimfiles/ftplugin/fortran/too.vim
```

The generic names for the filetype plugins are:

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

Here "<name>" can be any name that you prefer. Examples for the "stuff" filetype on Unix:

```
~/.vim/ftplugin/stuff.vim
~/.vim/ftplugin/stuff_def.vim
~/.vim/ftplugin/stuff/header.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`".

Further reading:
|`:h filetype-plugins`| Documentation for the filetype plugins and information about how to avoid that mappings cause problems.
|`:h load-plugins`| When the global plugins are loaded during startup.
|`:h ftplugin-overrule`| Overruling the settings from a global plugin.
|`write-plugin`| How to write a plugin script.

|:h plugin-details| For more information about using plugins or when your plugin doesn't work.
|:h new-filetype| How to detect a new file type.

Adding a help file

If you are lucky, the plugin you installed also comes with a help file. We will explain how to install the help file, so that you can easily find help for your new plugin. Let us use the "matchit.vim" plugin as an example (it is included with Vim). This plugin makes the "%" command jump to matching HTML tags, if/else/endif in Vim scripts, etc. Very useful, although it's not backwards compatible (that's why it is not enabled by default). This plugin comes with documentation: "matchit.txt". Let's first copy the plugin to the right directory. This time we will do it from inside Vim, so that we can use $VIMRUNTIME. (You may skip some of the "mkdir" commands if you already have the directory.)

```
:!mkdir ~/.vim
:!mkdir ~/.vim/plugin
:!cp $VIMRUNTIME/macros/matchit.vim ~/.vim/plugin
```

The "cp" command is for Unix, on MS-DOS you can use "copy".

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 |:h :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.

For writing a local help file, see |write-local-help|.

The option window

If you are looking for an option that does what you want, you can search in the help files here: |:h 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.

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&
```

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.

Wrapping movement commands

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,<,>,[,]
```

There are a few other flags that can be added, see 'whichwrap'.

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.

06. Using syntax highlighting

Black and white text is boring. With colors your file comes to life. This not only looks nice, it also speeds up your work. Change the colors used for the different sorts of text. Print your text, with the colors you see on the screen.

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 always want to use syntax highlighting, put the "`:syntax enable`" command in your |`:h vimrc`| file.

If you want syntax highlighting only when the terminal supports colors, you can put this in your |`:h 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 |`:h gvimrc`| file.

No or wrong colors?

There can be a number of reasons why you don't see colors:

Your terminal does not support colors. Vim will use bold, italic and underlined text, but this doesn't look very nice. You probably will want to try to get a terminal with colors. For Unix, I recommend the xterm from the XFree86 project: |`:h xfree-xterm`|.

Your terminal does support colors, but Vim doesn't know this. Make sure your `$TERM` setting is correct. For example, when using an xterm that supports colors:

```
setenv TERM xterm-color
```

or (depending on your shell):

```
TERM=xterm-color; export TERM
```

The terminal name must match the terminal you are using. If it still doesn't work, have a look at |`:h xterm-color`|, which shows a few ways to make Vim display colors (not only for an xterm).

The file type is not recognized. Vim doesn't know all file types, and sometimes it's near to impossible to tell what language a file uses. Try this command:

```
:set filetype
```

If the result is "filetype=" then the problem is indeed that Vim doesn't know what type of file this is. You can set the type manually:

```
:set filetype=fortran
```

To see which types are available, look in the directory `$VIMRUNTIME/syntax`. For the GUI you can use the Syntax menu. Setting the filetype can also be done with a |`:h modeline`|, so that the file will be highlighted each time you edit it. For example, this line can be used in a Makefile (put it near the start or end of the file):

```
# vim: syntax=make
```

You might know how to detect the file type yourself. Often the file name extension (after the dot) can be used. See |`:h new-filetype`| for how to tell Vim to detect that file type.

There is no highlighting for your file type. You could try using a similar file type by manually setting it as mentioned above. If that isn't good enough, you can write your own syntax file, see |`:h mysyntaxfile`|.

Or the colors could be wrong:

The colored text is very hard to read. Vim guesses the background color that you are using. If it is black (or another dark color) it will use light colors for text. If it is white (or another light color) it will use dark colors for text. If Vim guessed wrong the text will be hard to read. To solve this, set the `'background'` option. For a dark background:

```
:set background=dark
```

And for a light background:

```
:set background=light
```

Make sure you put this *before* the "`:syntax enable`" command, otherwise the colors will already have been set. You could do "`:syntax reset`" after setting `'background'` to make Vim set the default colors again.

The colors are wrong when scrolling bottom to top. Vim doesn't read the whole file to parse the text. It starts parsing wherever you are viewing the file. That saves a lot of time, but sometimes the colors are wrong. A simple fix is hitting CTRL-L. Or scroll back a bit and then forward again. For a real fix, see |`:h :syn-sync`|. Some syntax files have a way to make it look further back, see the help for the specific syntax file. For example, |`:h tex.vim`| for the TeX syntax.

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`.

When you found the color scheme that you like, add the "`:colorscheme`" command to your |`:h vimrc`| file.

You could also write your own color scheme. This is how you do it:

1. Select a color scheme that comes close. Copy this file to your own Vim directory. For Unix, this should work:

   ```
   !mkdir ~/.vim/colors
   !cp $VIMRUNTIME/colors/morning.vim ~/.vim/colors/mine.vim
   ```

   This is done from Vim, because it knows the value of `$VIMRUNTIME`.

2. Edit the color scheme file. These entries are useful:

term	attributes in a B&W terminal
cterm	attributes in a color terminal
ctermfg	foreground color in a color terminal
ctermbg	background color in a color terminal
gui	attributes in the GUI
guifg	foreground color in the GUI
guibg	background color in the GUI

   For example, to make comments green:

   ```
   :highlight Comment ctermfg=green guifg=green
   ```

   Attributes you can use for "`cterm`" and "`gui`" are "`bold`" and "`underline`". If you want both, use "`bold,underline`". For details see the |`:h :highlight`| command.

3. Tell Vim to always use your color scheme. Put this line in your |`:h vimrc`|:

   ```
   colorscheme mine
   ```

If you want to see what the most often used color combinations look like, use this command:

```
:runtime syntax/colortest.vim
```

You will see text in various color combinations. You can check which ones are readable and look nice.

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
```

Printing with colors

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

```
:hardcopy
```

You will get the usual printer dialog, where you can select the printer and a few settings. If you have a color printer, the paper output should look the same as what you see inside Vim. But when you use a dark background the colors will be adjusted to look good on white paper.

There are several options that change the way Vim prints:

- `'printdevice'`
- `'printheader'`
- `'printfont'`
- `'printoptions'`

To print only a range of lines, use Visual mode to select the lines and then type the command:

```
v100j:hardcopy
```

"`v`" starts Visual mode. "`100j`" moves a hundred lines down, they will be highlighted. Then "`:hardcopy`" will print those lines. You can use other commands to move in Visual mode, of course.

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
```

You will see it crunching away, this can take quite a while for a large file. Some time later another window shows the HTML code. Now write this somewhere (doesn't matter where, you throw it away later):

```
:write main.c.html
```

Open this file in your favorite browser and print it from there. If all goes well, the output should look exactly as it does in Vim. See |`:h 2html.vim`| for details. Don't forget to delete the HTML file when you are done with it.

Instead of printing, you could also put the HTML file on a web server, and let others look at the colored text.

Further reading

|`Your own syntax highlighted`|
|`:h syntax`| All the details.

07. More than one file

No matter how many files you have, you can edit them without leaving Vim. Define a list of files to work on and jump from one to the other. Copy text from one file and put it in another one.

Edit another file

So far you had to start Vim for every file you wanted to edit. There is a simpler way. To start editing another file, use this command:

```
:edit foo.txt
```

You can use any file name instead of "`foo.txt`". Vim will close the current file and open the new one. If the current file has unsaved changes, however, Vim displays an error message and does not open the new file:

```
E37: No write since last change (use ! to override)
```

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
```

At this point, you have a number of alternatives. You can write the file using this command:

```
:write
```

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
```

The text with changes is still there, but you can't see it. This is further explained in section |`The buffer list`|.

A list of files

You can start Vim to edit a sequence of files. For example:

```
vim one.c two.c three.c
```

This command starts Vim and tells it that you will be editing three files. Vim displays just the first file. After you have done your thing in this file, to edit the next file you use this command:

```
:next
```

If you have unsaved changes in the current file, you will get an error message and the "`:next`" will not work. This is the same problem as with "`:edit`" mentioned in the previous section. To abandon the changes:

```
:next!
```

But mostly you want to save the changes and move on to the next file. There is a special command for this:

```
:wnext
```

This does the same as using two separate commands:

```
:write
:next
```

WHERE AM I?

To see which file in the argument list you are editing, look in the window title. It should show something like "(2 of 3)". This means you are editing the second file out of three files.

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

```
:args
```

This is short for "arguments". The output might look like this:

```
    one.c [two.c] three.c
```

These are the files you started Vim with. The one you are currently editing, "`two.c`", is in square brackets.

Moving to other arguments

To go back one file:

```
:previous
```

This is just like the "`:next`" command, except that it moves in the other direction. Again, there is a shortcut command for when you want to write the file first:

```
:wprevious
```

To move to the very last file in the list:

```
:last
```

And to move back to the first one again:

```
:first
```

There is no "`:wlast`" or "`:wfirst`" command though!

You can use a count for "`:next`" and "`:previous`". To skip two files forward:

```
:2next
```

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
```

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.

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).

Jumping from file to file

To quickly jump between two files, press `CTRL-^` (on English-US keyboards the îs 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" file. When you just started Vim `CTRL-^` will not work, since there isn't a previous file.

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.

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.

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
```

The name of the backup file is the original file with a `~` added to the end. If your file is named `data.txt`, for example, the backup file name is `data.txt~`.

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

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

This will use `data.txt.bak` instead of `data.txt~`.

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).

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.

Copy text between files

This explains how to copy text from one file to another. Let's start with a simple example. Edit the file that contains the text you want to copy. Move the cursor to the start of the text and press "v". This starts Visual mode. Now move the cursor to the end of the text and press "y". This yanks (copies) the selected text.

To copy the above paragraph, you would do:

```
:edit thisfile
/This
vjjjj$y
```

Now edit the file you want to put the text in. Move the cursor to the character where you want the text to appear after. Use "p" to put the text there.

```
:edit otherfile
/There
p
```

Of course you can use many other commands to yank the text. For example, to select whole lines start Visual mode with "V". Or use CTRL-V to select a rectangular block. Or use "Y" to yank a single line, "yaw" to yank-a-word, etc.

The "p" command puts the text after the cursor. Use "P" to put the text before the cursor. Notice that Vim remembers if you yanked a whole line or a block, and puts it back that way.

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`".

Appending to a file

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

```
:write >> logfile
```

This will write the text of the current file to the end of "logfile". Thus it is appended. This avoids that you have to copy the lines, edit the log file and put them there. Thus you save two steps. But you can only append to the end of a file.

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.

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
```

You are now editing "file" in read-only mode. When you try using "`:w`" you will get an error message and the file won't be written.

When you try to make a change to the file Vim will give you a warning:

```
W10: Warning: Changing a readonly file
```

The change will be done though. This allows for formatting the file, for example, to be able to read it easily.

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
```

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.

08. Splitting windows

Display two different files above each other. Or view two locations in the file at the same time. See the difference between two files by putting them side by side. All this is possible with split windows.

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:

```
+----------------------------------+
|/* file one.c */                  |
|~                                 |
|~                                 |
|one.c=============================|
|/* file one.c */                  |
|~                                 |
|one.c=============================|
|                                  |
+----------------------------------+
```

What you see here is two windows on the same file. The line with "`====`" is that status line. It displays information about the window above it. (In practice the status line will be in reverse video.)

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.)

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.

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
```

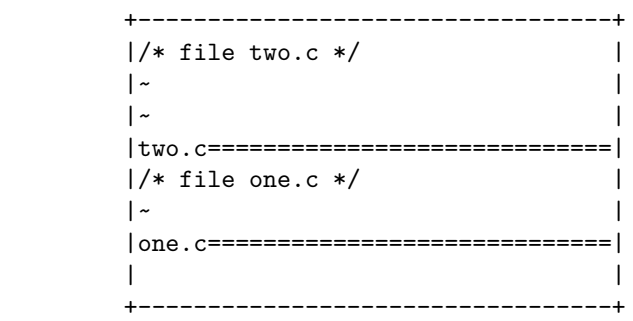
This closes all windows, except for the current one. If any of the other windows has changes, you will get an error message and that window won't be closed.

Split a window on another file

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

```
:split two.c
```

If you were editing `one.c`, then the result looks like this:



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

```
:new
```

You can repeat the "`:split`" and "`:new`" commands to create as many windows as you like.

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
```

For existing windows you can change the size in several ways. When you have a working mouse, it is easy: Move the mouse pointer to the status line that separates two windows, and drag it up or down.

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 _
```

That's: a number `{height},` CTRL-W and then an underscore (the - key with Shift on English-US keyboards).

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

Using the mouse

In Vim you can do many things very quickly from the keyboard. Unfortunately, the window resizing commands require quite a bit of typing. In this case, using the mouse is faster. Position the mouse pointer on a status line. Now press the left mouse button and drag. The status line will move, thus making the window on one side higher and the other smaller.

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.

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
```

The result looks something like this:

```
+--------------------------------------+
|/* file two.c */   |/* file one.c */  |
|~                  |~                 |
|~                  |~                 |
|~                  |~                 |
|two.c===============one.c=============|
|                                      |
+--------------------------------------+
```

Actually, the `|` lines in the middle will be in reverse video. This is called the vertical separator. It separates the two windows left and right of it.

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.)

Moving between windows

Since you can split windows horizontally and vertically as much as you like, you can create almost any layout of windows. Then you can use these commands to move between them:

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

You will notice the same letters as used for moving the cursor. And the cursor keys can also be used, if you like.

More commands to move to other windows: |`:h Q_wi`|.

Moving windows

You have split a few windows, but now they are in the wrong place. Then you need a command to move the window somewhere else. For example, you have three windows like this:

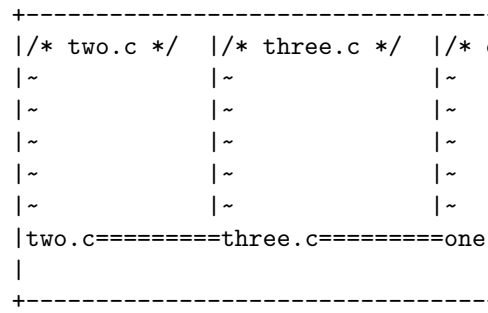
```
+----------------------------------+
|/* file two.c */                  |
|~                                 |
|~                                 |
|two.c=============================|
|/* file three.c */                |
|~                                 |
|~                                 |
|three.c===========================|
|/* file one.c */                  |
|~                                 |
|one.c=============================|
|                                  |
+----------------------------------+
```

Clearly the last one should be at the top. Go to that window (using CTRL-W w) and the type this command:

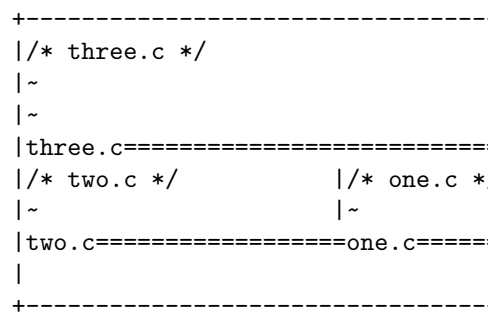
```
CTRL-W K
```

This uses the uppercase letter K. What happens is that the window is moved to the very top. You will notice that K is again used for moving upwards.

When you have vertical splits, CTRL-W K will move the current window to the top and make it occupy the full width of the Vim window. If this is your layout:



Then using CTRL-W K in the middle window (three.c) will result in:



The other three similar commands (you can probably guess these now):

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

Commands for all windows

When you have several windows open and you want to quit Vim, you can close each window separately. A quicker way is using this command:

```
:qall
```

This stands for "quit all". If any of the windows contain changes, Vim will not exit. The cursor will automatically be positioned in a window with changes. You can then either use "`:write`" to save the changes, or "`:quit!`" to throw them away.

If you know there are windows with changes, and you want to save all these changes, use this command:

```
:wall
```

This stands for "write all". But actually, it only writes files with changes. Vim knows it doesn't make sense to write files that were not changed.

And then there is the combination of "`:qall`" and "`:wall`": the "write and quit all" command:

```
:wqall
```

This writes all modified files and quits Vim.

Finally, there is a command that quits Vim and throws away all changes:

```
:qall!
```

Be careful, there is no way to undo this command!

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
```

This results in:

```
+-------------------------------+
|file one.txt                   |
|~                              |
|one.txt========================|
|file two.txt                   |
|~                              |
|two.txt========================|
|file three.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.

Viewing differences with vimdiff

There is a special way to start Vim, which shows the differences between two files. Let's take a file "main.c" and insert a few characters in one line. Write this file with the `'backup'` option set, so that the backup file "`main.c~`" will contain the previous version of the file.

Type this command in a shell (not in Vim):

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

Vim will start, with two windows side by side. You will only see the line in which you added characters, and a few lines above and below it.

```
 VV           VV
+----------------------------------------+
|+ +--123 lines: /* a|+ +--123 lines: /* a|  <- fold
|  text              |  text              |
|  text              |  text              |
|  text              |  text              |
|  text              |  changed text      |  <- changed line
|  text              |  text              |
|  text              |  ------------------|  <- deleted line
|  text              |  text              |
|  text              |  text              |
|  text              |  text              |
|+ +--432 lines: text|+ +--432 lines: text|  <- fold
|  ~                 | ~                  |
|  ~                 | ~                  |
|main.c~==============main.c==============|
|                                        |
+----------------------------------------+
```

(This picture doesn't show the highlighting, use the vimdiff command for a better look.)

The lines that were not modified have been collapsed into one line. This is called a closed fold. They are indicated in the picture with "`<- fold`". Thus the single fold line at the top stands for 123 text lines. These lines are equal in both files.

The line marked with "`<- changed line`" is highlighted, and the inserted text is displayed with another color. This clearly shows what the difference is between the two files.

The line that was deleted is displayed with "`---`" in the main.c window. See the "`<- deleted line`" marker in the picture. These characters are not really there. They just fill up `main.c`, so that it displays the same number of lines as the other window.

The fold column

Each window has a column on the left with a slightly different background. In the picture above these are indicated with "`VV`". You notice there is a plus character there, in front of each closed fold. Move the mouse pointer to that plus and click the left button. The fold will open, and you can see the text that it contains.

The fold column contains a minus sign for an open fold. If you click on this `-`, the fold will close.

Obviously, this only works when you have a working mouse. You can also use "`zo`" to open a fold and "`zc`" to close it.

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~
```

The "`:vertical`" command is used to make the window split vertically. If you omit this, you will get a horizontal split.

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).

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
```

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.

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 |`:h vimdiff`|.

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

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

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
```

This will edit the file "thatfile" in a window that occupies the whole Vim window. And you will notice a bar at the top with the two file names:

```
+----------------------------------+
| thisfile | /thatfile/ __________X|    (thatfile is bold)
|/* thatfile */                    |
|that                              |
|that                              |
|~                                 |
|~                                 |
|~                                 |
|                                  |
+----------------------------------+
```

You now have two tab pages. The first one has a window for "thisfile" and the second one a window for "thatfile". It's like two pages that are on top of eachother, with a tab sticking out of each page showing the file name.

Now use the mouse to click on "thisfile" in the top line. The result is

```
+----------------------------------+
| /thisfile/ | thatfile __________X|    (thisfile is bold)
|/* thisfile */                    |
|this                              |
|this                              |
|~                                 |
|~                                 |
|~                                 |
|                                  |
+----------------------------------+
```

Thus you can switch between tab pages by clicking on the label in the top line. 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
```

This makes a new tab page with one window that is editing the same buffer as the window we were in:

```
+--------------------------------------+
| thisfile | /thisfile/ | thatfile __X|   (thisfile is bold)
|/* thisfile */                       |
|this                                 |
|this                                 |
|~                                    |
|~                                    |
|~                                    |
|                                     |
+--------------------------------------+
```

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
```

Will show the help text for "`gt`" in a new tab page.

A few more things you can do with tab pages:

ce after the last label The next tab page will be selected, like with "gt".

n the top right corner	The current tab page will be closed. Unless there are unsaved changes in the current tab page.
mouse in the top line	A new tab page will be created.
”tabonly” command	Closes all tab pages except the current one. Unless there are unsaved changes in other tab pages.

For more information about tab pages see |`:h tab-page`|.

09. 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.

Parts of the GUI

You might have an icon on your desktop that starts gVim. Otherwise, one of these commands should do it:

```
gvim file.txt
vim -g file.txt
```

If this doesn't work you don't have a version of Vim with GUI support. You will have to install one first.

Vim will open a window and display "`file.txt`" in it. What the window looks like depends on the version of Vim. It should resemble the following picture (for as far as this can be shown in ASCII!).

```
+-----------------------------------------------------+
| file.txt + (~/dir) - VIM                         X  | <- window title
+-----------------------------------------------------+
| File Edit  Tools  Syntax  Buffers  Window  Help     | <- menubar
+-----------------------------------------------------+
| aaa  bbb  ccc  ddd  eee  fff ggg  hhh  iii  jjj     | <- toolbar
| aaa  bbb  ccc  ddd  eee  fff ggg  hhh  iii  jjj     |
+-----------------------------------------------------+
| file text                                        | ^ |
| ~                                                | # |
| ~                                                | # | <- scrollbar
| ~                                                | # |
| ~                                                | # |
| ~                                                | # |
|                                                  | V |
+-----------------------------------------------------+
```

The largest space is occupied by the file text. This shows the file in the same way as in a terminal. With some different colors and another font perhaps.

The window title

At the very top is the window title. This is drawn by your window system. Vim will set the title to show the name of the current file. First comes the name of the file. Then some special characters and the directory of the file in parens. These special character can be present:

`-`	The file cannot be modified (e.g., a help file)
`+`	The file contains changes
`=`	The file is read-only
`=+`	The file is read-only, contains changes anyway

If nothing is shown you have an ordinary, unchanged file.

The menubar

You know how menus work, right? Vim has the usual items, plus a few more. Browse them to get an idea of what you can use them for. A relevant submenu is Edit/Global Settings. You will find these entries:

Toggle Toolbar	make the toolbar appear/disappear
Toggle Bottom Scrollbar	make a scrollbar appear/disappear at the bottom
Toggle Left Scrollbar	make a scrollbar appear/disappear at the left
Toggle Right Scrollbar	make a scrollbar appear/disappear at the right

On most systems you can tear-off the menus. Select the top item of the menu, the one that looks like a dashed line. You will get a separate window with the items of the menu. It will hang around until you close the window.

The toolbar

This contains icons for the most often used actions. Hopefully the icons are self-explanatory. There are tooltips to get an extra hint (move the mouse pointer to the icon without clicking and don't move it for a second).

The "Edit/Global Settings/Toggle Toolbar" menu item can be used to make the toolbar disappear. If you never want a toolbar, use this command in your vimrc file:

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

This removes the `'T'` flag from the `'guioptions'` option. Other parts of the GUI can also be enabled or disabled with this option. See the help for it.

The scrollbars

By default there is one scrollbar on the right. It does the obvious thing. When you split the window, each window will get its own scrollbar.

You can make a horizontal scrollbar appear with the menu item Edit/Global Settings/Toggle Bottom Scrollbar. This is useful in diff mode, or when the `'wrap'` option has been reset (more about that later).

When there are vertically split windows, only the windows on the right side will have a scrollbar. However, when you move the cursor to a window on the left, it will be this one the that scrollbar controls. This takes a bit of time to get used to.

When you work with vertically split windows, consider adding a scrollbar on the left. This can be done with a menu item, or with the `'guioptions'` option:

```
:set guioptions+=l
```

This adds the `'l'` flag to `'guioptions'`.

Using the mouse

Standards are wonderful. In Microsoft Windows, you can use the mouse to select text in a standard manner. The X Window system also has a standard system for using the mouse. Unfortunately, these two standards are not the same.

Fortunately, you can customize Vim. You can make the behavior of the mouse work like an X Window system mouse or a Microsoft Windows mouse. The following command makes the mouse behave like an X Window mouse:

```
:behave xterm
```

The following command makes the mouse work like a Microsoft Windows mouse:

```
:behave mswin
```

The default behavior of the mouse on UNIX systems is xterm. The default behavior on a Microsoft Windows system is selected during the installation process. For details about what the two behaviors are, see |`:h :behave`|. Here follows a summary.

Xterm mouse behavior

Left mouse click	position the cursor
Left mouse drag	select text in Visual mode
Middle mouse click	paste text from the clipboard
Right mouse click	extend the selected text until the mouse pointer

Mswin mouse behavior

Left mouse click	position the cursor
Left mouse drag	select text in Select mode (see |`Select mode`|)
Left mouse click, with Shift	extend the selected text until the mouse pointer
Middle mouse click	paste text from the clipboard
Right mouse click	display a pop-up menu

The mouse can be further tuned. Check out these options if you want to change the way how the mouse works:

`'mouse'`	in which mode the mouse is used by Vim
`'mousemodel'`	what effect a mouse click has
`'mousetime'`	time between clicks for a double-click
`'mousehide'`	hide the mouse while typing
`'selectmode'`	whether the mouse starts Visual or Select mode

The clipboard

In section |`Using the clipboard`| the basic use of the clipboard was explained. There is one essential thing to explain about X-windows: There are actually two places to exchange text between programs. MS-Windows doesn't have this.

In X-Windows there is the "current selection". This is the text that is currently highlighted. In Vim this is the Visual area (this assumes you are using the default option settings). You can paste this selection in another application without any further action.

For example, in this text select a few words with the mouse. Vim will switch to Visual mode and highlight the text. Now start another gVim, without a file name argument, so that it displays an empty window. Click the middle mouse button. The selected text will be inserted.

The "current selection" will only remain valid until some other text is selected. After doing the paste in the other gVim, now select some characters in that window. You will notice that the words that were previously selected in the other gVim window are displayed differently. This means that it no longer is the current selection.

You don't need to select text with the mouse, using the keyboard commands for Visual mode works just as well.

The real clipboard

Now for the other place with which text can be exchanged. We call this the "real clipboard", to avoid confusion. Often both the "current selection" and the "real clipboard" are called clipboard, you'll have to get used to that.

To put text on the real clipboard, select a few different words in one of the gVims you have running. Then use the Edit/Copy menu entry. Now the text has been copied to the real clipboard. You can't see this, unless you have some application that shows the clipboard contents (e.g., KDE's klipper).

Now select the other gVim, position the cursor somewhere and use the Edit/Paste menu. You will see the text from the real clipboard is inserted.

Using both

This use of both the "current selection" and the "real clipboard" might sound a bit confusing. But it is very useful. Let's show this with an example. Use one gVim with a text file and perform these actions:

- Select two words in Visual mode.
- Use the Edit/Copy menu to get these words onto the clipboard.
- Select one other word in Visual mode.
- Use the Edit/Paste menu item. What will happen is that the single selected word is replaced with the two words from the clipboard.
- Move the mouse pointer somewhere else and click the middle button. You will see that the word you just overwrote with the clipboard is inserted here.

If you use the "current selection" and the "real clipboard" with care, you can do a lot of useful editing with them.

Using the keyboard

If you don't like using the mouse, you can access the current selection and the real clipboard with two registers. The `"*` register is for the current selection.

To make text become the current selection, use Visual mode. For example, to select a whole line just press "V".

To insert the current selection before the cursor:

```
"*P
```

Notice the uppercase "P". The lowercase "p" puts the text after the cursor.

The "+ register is used for the real clipboard. For example, to copy the text from the cursor position until the end of the line to the clipboard:

```
"+y$
```

Remember, "y" is yank, which is Vim's copy command.

To insert the contents of the real clipboard before the cursor:

```
"+P
```

It's the same as for the current selection, but uses the plus (+) register instead of the star (*) register.

Select mode

And now something that is used more often on MS-Windows than on X-Windows. But both can do it. You already know about Visual mode. Select mode is like Visual mode, because it is also used to select text. But there is an obvious difference: When typing text, the selected text is deleted and the typed text replaces it.

To start working with Select mode, you must first enable it (for MS-Windows it is probably already enabled, but you can do this anyway):

```
:set selectmode+=mouse
```

Now use the mouse to select some text. It is highlighted like in Visual mode. Now press a letter. The selected text is deleted, and the single letter replaces it. You are in Insert mode now, thus you can continue typing.

Since typing normal text causes the selected text to be deleted, you can not use the normal movement commands "hjkl", "w", etc. Instead, use the shifted function keys. <S-Left> (shifted cursor left key) moves the cursor left. The selected text is changed like in Visual mode. The other shifted cursor keys do what you expect. <S-End> and <S-Home> also work.

You can tune the way Select mode works with the 'selectmode' option.

10. 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.

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 can now execute the macro by typing the command "`@{register}`".

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"
```

You start by moving to the first character of the first line. Next you execute the following commands:

`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
```

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.

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.

Now you can execute the corrected commands with "@n". (If your recorded commands include line breaks, adjust the last two items in the example to include all the lines.)

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.

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/
```

Note: The "`:substitute`" command is almost never spelled out completely. Most of the time, people use the abbreviated version "`:s`". From here on the abbreviation will be used.

The "`%`" before the command specifies the command works on all lines. Without a range, "`:s`" only works on the current line. More about ranges in the next section |`Command ranges`|.

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
```

results in (starting with the original line):

```
    Teacher Smith criticized Teacher Johnson today.
```

Other flags include `p` (print), which causes the "`:substitute`" command to print out the last line it changes. The `c` (confirm) flag tells "`:substitute`" to ask you for confirmation before it performs each substitution. Enter the following:

```
:%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 or two+
```

Command ranges

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

The simple form of a range is {number}, {number}. For example:

```
:1,5s/this/that/g
```

Executes the substitute command on the lines 1 to 5. Line 5 is included. The range is always placed before the command.

A single number can be used to address one specific line:

```
:54s/President/Fool/
```

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. The ":write" command works like that. Without a range, it writes the whole file. To make it write only the current line into a file:

```
:.write otherfile
```

The first line always has number one. How about the last line? The "$" character is used for this. For example, to substitute in the lines from the cursor to the end:

```
:.,$s/yes/no/
```

The "%" range that we used before, is actually a short way to say "1,$", from the first to the last line.

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.

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.

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
```

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:

```
:'>,$
```

This addresses the lines from the end of the Visual area to the end of the file.

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.

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.

Note: The command executed for ":global" must be one that starts with a colon. Normal mode commands can not be used directly. The |:h :normal| command can do this for you.

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

With CTRL-V you can start selection of a rectangular area of text. There are a few commands that do something special with the text block.

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.

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. Example:

```
include one
include two
include three
include four
```

Move the cursor to the "o" of "one" and press CTRL-V. Move it down with "3j" to "four". You now have a block selection that spans four lines. Now type:

```
Imain.<Esc>
```

The result:

```
include main.one
include main.two
include main.three
include main.four
```

If the block spans short lines that do not extend into the block, the text is not inserted in that line. For example, make a Visual block selection that includes the word "long" in the first and last line of this text, and thus has no text selected in the second line:

```
This is a long line
short
Any other long line

          ^^^^ selected block
```

Now use the command "`Ivery <Esc>`". The result is:

```
This is a very long line
short
Any other very long line
```

In the short line no text was inserted.

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. And it does insert text in a short line. Thus you can make a choice whether you do or don't want to append text to a short line.

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.

Using the same example from above, and then typing "`$A XXX<Esc>`", you get this result:

```
This is a long line XXX
short XXX
Any other long line XXX
```

This really requires using the "`$`" command. Vim remembers that it was used. Making the same selection by moving the cursor to the end of the longest line with other movement commands will not have the same result.

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.

Starting with the same selection of the "long" words as above, then typing "`c_LONG_<Esc>`", you get this:

```
This is a _LONG_ line
short
Any other _LONG_ line
```

Just like with "`I`" the short line is not changed. Also, you can't enter a newline in the new text.

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.

Starting with the same text again, and typing "`Cnew text<Esc>`" you get:

```
This is a new text
short
Any other new text
```

Notice that, even though only the "long" word was selected, the text after it is deleted as well. Thus only the location of the left edge of the visual block really matters.

Again, short lines that do not reach into the block are excluded.

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)

Filling with a character

To fill the whole block with one character, use the "`r`" command. Again, starting with the same example text from above, and then typing "`rx`":

```
This is a xxxx line
short
Any other xxxx line
```

Note: If you want to include characters beyond the end of the line in the block, check out the `'virtualedit'` feature in chapter 25.

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.

With the same example again, "`>`" gives this result:

```
This is a     long line
short
Any other     long line
```

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.

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).

Let's use the example that we got so familiar with now. The result of using the "`J`" command:

```
This is a long line short Any other long line
```

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.

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.

Starting with this text:

```
Hi John,
Here is the diff that fixes the bug:
Bye, Pierre.
```

Move the cursor to the second line and type:

```
:read patch
```

The file named "patch" will be inserted, with this result:

```
Hi John,
Here is the diff that fixes the bug:
2c2
<    for (i = 0; i <= length; ++i)
---
>    for (i = 0; i < length; ++i)
Bye, Pierre.
```

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
```

The file "patch" will be put above the first line of the file.

Writing a range of lines

To write a range of lines to a file, the ":write" command can be used. Without a range it writes the whole file. With a range only the specified lines are written:

```
:.,$write tempo
```

This writes the lines from the cursor until the end of the file into the file "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 !:

```
:.,$write! tempo
```

CAREFUL: The ! must follow the ":write" command immediately, without white space. Otherwise it becomes a filter command, which is explained later in this chapter.

Appending to a file

In the first section of this chapter was explained how to collect a number of lines into a register. The same can be done to collect lines in a file. Write the first line with this command:

```
:.write collection
```

Now move the cursor to the second line you want to collect, and type this:

```
:.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

When you are typing plain text, it's nice if the length of each line is automatically trimmed to fit in the window. To make this happen while inserting text, set the `'textwidth'` option:

```
:set textwidth=72
```

You might remember that in the example vimrc file this command was used for every text file. Thus if you are using that vimrc file, you were already using it. To check the current value of `'textwidth'`:

```
:set textwidth
```

Now lines will be broken to take only up to 72 characters. But when you insert text halfway a line, or when you delete a few words, the lines will get too long or too short. Vim doesn't automatically reformat the text.

To tell Vim to format the current paragraph:

```
gqap
```

This starts with the "`gq`" command, which is an operator. Following is "`ap`", the text object that stands for "a paragraph". A paragraph is separated from the next paragraph by an empty line.

Note: A blank line, which contains white space, does NOT separate paragraphs. This is hard to notice!

Instead of "`ap`" you could use any motion or text object. If your paragraphs are properly separated, you can use this command to format the whole file:

```
gggqG
```

"`gg`" takes you to the first line, "`gq`" is the format operator and "`G`" the motion that jumps to the last line.

In case your paragraphs aren't clearly defined, you can format just the lines you manually select. Move the cursor to the first line you want to format. Start with the command "`gqj`". This formats the current line and the one below it. If the first line was short, words from the next line will be appended. If it was too long, words will be moved to the next line. The cursor moves to the second line. Now you can use "`.`" to repeat the command. Keep doing this until you are at the end of the text you want to format.

Changing case

You have text with section headers in lowercase. You want to make the word "section" all uppercase. Do this with the "`gU`" operator. Start with the cursor in the first column:

```
                       gUw
   section header      ---->      SECTION header
```

The "gu" operator does exactly the opposite:

```
                        guw
    SECTION header      ---->       section header
```

You can also use "g~" to swap case. 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~~". Example:

```
                             g~~
    Some GIRLS have Fun      ---->   sOME girls HAVE fUN
```

Using an external program

Vim has a very powerful set of commands, it can do anything. But there may still be something that an external command can do better or faster.

The command "!{motion}{program}" takes a block of text and filters it through an external program. In other words, it runs the system command represented by {program}, giving it the block of text represented by {motion} as input. The output of this command then replaces the selected block.

Because this summarizes badly if you are unfamiliar with UNIX filters, take a look at an example. The sort command sorts a file. If you execute the following command, the unsorted file input.txt will be sorted and written to output.txt. (This works on both UNIX and Microsoft Windows.)

```
sort <input.txt >output.txt
```

Now do the same thing in Vim. You want to sort lines 1 through 5 of a file. You start by putting the cursor on line 1. Next you execute the following command:

```
!5G
```

The "!" tells Vim that you are performing a filter operation. The Vim editor expects a motion command to follow, indicating which part of the file to filter. The "5G" command tells Vim to go to line 5, so it now knows that it is to filter lines 1 (the current line) through 5.

In anticipation of the filtering, the cursor drops to the bottom of the screen and a ! prompt displays. You can now type in the name of the filter program, in this case "sort". Therefore, your full command is as follows:

```
!5Gsort<Enter>
```

The result is that the sort program is run on the first 5 lines. The output of the program replaces these lines.

```
    line 55               line 11
    line 33               line 22
    line 11     -->       line 33
    line 22               line 44
    line 44               line 55
    last line             last line
```

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.

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.

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
```

This inserts the current time and date in UTC format at the top of the file. (Well, if you have a date command that accepts the "`-u`" argument.) Note the difference with using "!!date": that replaced a line, while ":read !date" will insert a line.

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. The output could look like this:

```
       4      47     249
```

The "`wc`" command isn't verbose. This means you have 4 lines, 47 words and 249 characters.

Watch out for this mistake:

```
:write! wc
```

This will write the file "wc" in the current directory, with force. White space is important here!

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
```

11. 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.

Basic recovery

In most cases recovering a file is quite simple, assuming you know which file you were editing (and the harddisk is still working). Start Vim on the file, with the "`-r`" argument added:

```
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. If Vim recovered your changes you will see these messages (with different file names, of course):

```
Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. You should check if everything is OK.
(You might want to write out this file under another name
and run diff with the original file to check for changes)
You may want to delete the .swp file now.
```

To be on the safe side, write this file under another name:

```
:write help.txt.recovered
```

Compare the file with the original file to check if you ended up with what you expected. Vimdiff is very useful for this |`Viewing differences with vimdiff`|. For example:

```
:write help.txt.recovered
:edit #
:diffsp help.txt
```

Watch out for the original file to contain a more recent version (you saved the file just before the computer crashed). And check that no lines are missing (something went wrong that Vim could not recover). If Vim produces warning messages when recovering, read them carefully. This is rare though.

If the recovery resulted in text that is exactly the same as the file contents, you will get this message:

```
Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. Buffer contents equals file contents.
You may want to delete the .swp file now.
```

This usually happens if you already recovered your changes, or you wrote the file after making changes. It is safe to delete the swap file now.

It is normal that the last few changes can not be recovered. Vim flushes the changes to disk when you don't type for about four seconds, or after typing about two hundred characters. This is set with the `'updatetime'` and `'updatecount'` options. Thus when Vim didn't get a chance to save itself when the system went down, the changes after the last flush will be lost.

If you were editing without a file name, give an empty string as argument:

```
vim -r ""
```

You must be in the right directory, otherwise Vim can't find the swap file.

Where is the swap file?

Vim can store the swap file in several places. Normally it is in the same directory as the original file. To find it, change to the directory of the file, and use:

```
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. The output could look like this:

```
Swap files found:
   In current directory:
1.    .main.c.swp
      owned by: mool   dated: Tue May 29 21:00:25 2001
     file name: ~mool/vim/vim6/src/main.c
      modified: YES
     user name: mool   host name: masaka.moolenaar.net
    process ID: 12525
   In directory ~/tmp:
      -- none --
   In directory /var/tmp:
      -- none --
   In directory /tmp:
      -- none --
```

If there are several swap files that look like they may be the one you want to use, a list is given of these swap files and you are requested to enter the number of the one you want to use. Carefully look at the dates to decide which one you want to use. In case you don't know which one to use, just try them one by one and check the resulting files if they are what you expected.

Using a specific swap file

If you know which swap file needs to be used, you can recover by giving the swap file name. Vim will then finds out the name of the original file from the swap file.

Example:

```
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 the 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.

Crashed or not?

Vim tries to protect you from doing stupid things. Suppose you innocently start editing a file, expecting the contents of the file to show up. Instead, Vim produces a very long message:

```
    E325: ATTENTION
Found a swap file by the name ".main.c.swp"
      owned by: mool   dated: Tue May 29 21:09:28 2001
     file name: ~mool/vim/vim6/src/main.c
      modified: no
     user name: mool   host name: masaka.moolenaar.net
    process ID: 12559 (still running)
While opening file "main.c"
         dated: Tue May 29 19:46:12 2001

(1) Another program may be editing the same file.
    If this is the case, be careful not to end up with two
    different instances of the same file when making changes.
    Quit, or continue with caution.

(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r main.c"
    to recover the changes (see ":help recovery").
    If you did this already, delete the swap file ".main.c.swp"
    to avoid this message.
```

You get this message, because, when starting to edit a file, Vim checks if a swap file already exists for that file. If there is one, there must be something wrong. It may be one of these two situations.

1. Another edit session is active on this file. Look in the message for the line with "process ID". It might look like this:

   ```
   process ID: 12559 (still running)
   ```

   The text "(still running)" indicates that the process editing this file runs on the same computer. When working on a non-Unix system you will not get this extra hint. When editing a file over a network, you may not see the hint, because the process might be running on another computer. In those two cases you must find out what the situation is yourself. If there is another Vim editing the same file, continuing to edit will result in two versions of the same file. The one that is written last will overwrite the other one, resulting in loss of changes. You better quit this Vim.

2. The swap file might be the result from a previous crash of Vim or the computer. Check the dates mentioned in the message. If the date of the swap file is newer than the file you were editing, and this line appears:

   ```
   modified: YES
   ```

   Then you very likely have a crashed edit session that is worth recovering. If the date of the file is newer than the date of the swap file, then either it was changed after the crash (perhaps you recovered it earlier, but didn't delete the swap file?), or else the file was saved before the crash but after the last write of the swap file (then you're lucky: you don't even need that old swap file). Vim will warn you for this with this extra line:

   ```
   NEWER than swap file!
   ```

Unreadable swap file

Sometimes the line

```
[cannot be read]
```

will appear under the name of the swap file. This can be good or bad, depending on circumstances.

It is good if a previous editing session crashed without having made any changes to the file. Then a directory listing of the swap file will show that it has zero bytes. You may delete it and proceed.

It is slightly bad if you don't have read permission for the swap file. You may want to view the file read-only, or quit. On multi-user systems, if you yourself did the last changes under a different login name, a logout followed by a login under that other name might cure the "read error". Or else you might want to find out who last edited (or is editing) the file and have a talk with them.

It is very bad if it means there is a physical read error on the disk containing the swap file. Fortunately, this almost never happens. You may want to view the file read-only at first (if you can), to see the extent of the changes that were "forgotten". If you are the one in charge of that file, be prepared to redo your last changes.

What to do?

If dialogs are supported you will be asked to select one of five choices:

```
Swap file ".main.c.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort, (D)elete it:
```

- O Open the file readonly. Use this when you just want to view the file and don't need to recover it. You might want to use this when you know someone else is editing the file, but you just want to look in it and not make changes.
- E Edit the file anyway. Use this with caution! If the file is being edited in another Vim, you might end up with two versions of the file. Vim will try to warn you when this happens, but better be safe then sorry.
- R Recover the file from the swap file. Use this if you know that the swap file contains changes that you want to recover.
- Q Quit. This avoids starting to edit the file. Use this if there is another Vim editing the same file. When you just started Vim, this will exit Vim. When starting Vim with files in several windows, Vim quits only if there is a swap file for the first one. When using an edit command, the file will not be loaded and you are taken back to the previously edited file.
- A Abort. Like Quit, but also abort further commands. This is useful when loading a script that edits several files, such as a session with multiple windows.
- D Delete the swap file. Use this when you are sure you no longer need it. For example, when it doesn't contain changes, or when the file itself is newer than the swap file. On Unix this choice is only offered when the process that created the swap file does not appear to be running.

If you do not get the dialog (you are running a version of Vim that does not support it), you will have to do it manually. To recover the file, use this command:

```
:recover
```

Vim cannot always detect that a swap file already exists for a file. This is the case when the other edit session puts the swap files in another directory or when the path name for the file is different when editing it on different machines. Therefore, don't rely on Vim always warning you.

If you really don't want to see this message, you can add the 'A' flag to the 'shortmess' option. But it's very unusual that you need this.

For remarks about encryption and the swap file, see |:h :recover-crypt|.

Further reading

|:h swap-file| An explanation about where the swap file will be created and what its name is.
|:h :preserve| Manually flushing the swap file to disk.
|:h :swapname| See the name of the swap file for the current file.
'updatecount' Number of key strokes after which the swap file is flushed to disk.
'updatetime' Timeout after which the swap file is flushed to disk.
'swapsync' Whether the disk is synced when the swap file is flushed.
'directory' List of directory names where to store the swap file.
'maxmem' Limit for memory usage before writing text to the swap file.
'maxmemtot' Same, but for all files in total.

12. 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.

Replace a word

The substitute command can be used to replace all occurrences of a word with another word:

```
:%s/four/4/g
```

The "`%`" range means to replace in all lines. The "`g`" flag at the end causes all words in a line to be replaced.

This will not do the right thing if your file also contains "thirtyfour". It would be replaced with "thirty4". To avoid this, use the "`\<`" item to match the start of a word:

```
:%s/\<four/4/g
```

Obviously, this still goes wrong on "fourteen". Use "`\>`" to match the end of a word:

```
:%s/\<four\>/4/g
```

If you are programming, you might want to replace "four" in comments, but not in the code. Since this is difficult to specify, add the "`c`" flag to have the substitute command prompt you for each replacement:

```
:%s/\<four\>/4/gc
```

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.

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/
```

Let's break this down in parts. Obviously it starts with a substitute command. The "%" is the line range, which stands for the whole file. Thus the substitution is done in every line in the file.

The arguments for the substitute command are "/from/to/". The slashes separate the "from" pattern and the "to" string. 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 |:h sub-replace-special|.

Sort a list

In a Makefile you often have a list of files. For example:

```
OBJS = \
    version.o \
    pch.o \
    getopt.o \
    util.o \
    getopt1.o \
    inp.o \
    patch.o \
    backup.o
```

To sort this list, filter the text through the external sort command:

```
/^OBJS
j
:.,/^$/-1!sort
```

This goes to the first line, where "`OBJS`" is the first thing in the line. Then it goes one line down and filters the lines until the next empty line. You could also select the lines in Visual mode and then use "`!sort`". That's easier to type, but more work when there are many lines.

The result is this:

```
OBJS = \
    backup.o
    getopt.o \
    getopt1.o \
    inp.o \
    patch.o \
    pch.o \
    util.o \
    version.o \
```

Notice that a backslash at the end of each line is used to indicate the line continues. After sorting, this is wrong! The "`backup.o`" line that was at the end didn't have a backslash. Now that it sorts to another place, it must have a backslash.

The simplest solution is to add the backslash with "`A \<Esc>`". You can keep the backslash in the last line, if you make sure an empty line comes after it. That way you don't have this problem again.

Reverse line order

The |:global| command can be combined with the |`:h :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 |`:h :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, |`:h :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
```

Count words

Sometimes you have to write a text with a maximum number of words. Vim can count the words for you.

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

```
g CTRL-G
```

Do not type a space after the g, this is just used here to make the command easy to read.

The output looks like this:

```
Col 1 of 0; Line 141 of 157; Word 748 of 774; Byte 4489 of 4976
```

You can see on which word you are (748), and the total number of words in the file (774).

When the text is only part of a file, you could move to the start of the text, type "`g CTRL-G`", move to the end of the text, type "g CTRL-G" again, and then use your brain to compute the difference in the word position. That's a good exercise, but there is an easier way. With Visual mode, select the text you want to count words in. Then type `g CTRL-G`. The result:

```
Selected 5 of 293 Lines; 70 of 1884 Words; 359 of 10928 Bytes
```

For other ways to count words, lines and other items, see |`:h count-items`|.

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 to the word you want to find help on and press

```
K
```

Vim will run the external "`man`" program on the word. If the man page is found, it is displayed. This uses the normal pager to scroll through the text (mostly the "`more`" program). When you get to the end pressing <Enter> will get you back into Vim.

A disadvantage is that you can't see the man page and the text you are working on at the same time. There is a trick to make the man page appear in a Vim window. First, load the man filetype plugin:

```
:runtime! ftplugin/man.vim
```

Put this command in your vimrc file if you intend to do this often. Now you can use the "`:Man`" command to open a window on a man page:

```
:Man csh
```

You can scroll around and the text is highlighted. This allows you to find the help you were looking for. Use `CTRL-W w` to jump to the window with the text you were working on.

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

```
:Man 3 echo
```

To jump to another man page, which is in the text with the typical form "`word(1)`", press CTRL-] on it. Further "`:Man`" commands will use the same window.

To display a man page for the word under the cursor, use this:

```
\K
```

(If you redefined the <Leader>, use it instead of the backslash). For example, you want to know the return value of "`strstr()`" while editing this line:

```
    if ( strstr (input, "aap") == )
```

Move the cursor to somewhere on "`strstr`" and type "`\K`". A window will open to display the man page for strstr().

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\+$//
```

The line range "`%`" is used, thus this works on the whole file. The pattern that the "`:substitute`" command matches with is "`\s\+$`". This finds white space characters (`\s`), 1 or more of them (`\+`), before the end-of-line (`$`). Later will be explained how you write patterns like this |`Search commands and patterns`|.

The "to" part of the substitute command is empty: "`//`". Thus it replaces with nothing, effectively deleting the matched white space.

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.

Find where a word is used

If you are a UNIX user, you can use a combination of Vim and the grep command to edit all the files that contain a given word. This is extremely useful if you are working on a program and want to view or edit all the files that contain a specific variable.

For example, suppose you want to edit all the C program files that contain the word "`frame_counter`". To do this you use the command:

```
vim `grep -l frame_counter *.c`
```

Let's look at this command in detail. The grep command searches through a set of files for a given word. Because the -l argument is specified, the command will only list the files containing the word and not print the matching lines. The word it is searching for is "`frame_counter`". Actually, this can be any regular expression. (Note: What grep uses for regular expressions is not exactly the same as what Vim uses.)

The entire command is enclosed in backticks (`` ` ``). This tells the UNIX shell to run this command and pretend that the results were typed on the command line. So what happens is that the grep command is run and

produces a list of files, these files are put on the Vim command line. This results in Vim editing the file list that is the output of grep. You can then use commands like ":next" and ":first" to browse through the files.

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 next matching line (no matter in what file it is), use the "`:cnext`" command. To go to 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'`.

20. Typing command-line commands quickly

Vim has a few generic features that makes it easier to enter commands. Colon commands can be abbreviated, edited and repeated. Completion is available for nearly everything.

Command line editing

When you use a colon (`:`) command or search for a string with `/` or `?`, Vim puts the cursor on the bottom of the screen. There you type the command or search pattern. This is called the Command line. Also when it's used for entering a search command.

The most obvious way to edit the command you type is by pressing the <BS> key. This erases the character before the cursor. To erase another character, typed earlier, first move the cursor with the cursor keys.

For example, you have typed this:

```
:s/col/pig/
```

Before you hit <Enter>, you notice that "col" should be "cow". To correct this, you type <Left> five times. The cursor is now just after "col". Type <BS> and "`w`" to correct:

```
:s/cow/pig/
```

Now you can press <Enter> directly. You don't have to move the cursor to the end of the line before executing the command.

The most often used keys to move around in the command line:

<Left>	one character left
<Right>	one character right
<S-Left> or <C-Left>	one word left
<S-Right> or <C-Right>	one word right
CTRL-B or <Home>	to begin of command line
CTRL-E or <End>	to end of command line

Note: <S-Left> (cursor left key with Shift key pressed) and <C-Left> (cursor left key with Control pressed) will not work on all keyboards. Same for the other Shift and Control combinations.

You can also use the mouse to move the cursor.

Deleting

As mentioned, <BS> deletes the character before the cursor. To delete a whole word use CTRL-W.

```
/the fine pig

        CTRL-W

/the fine
```

CTRL-U removes all text, thus allows you to start all over again.

Overstrike

The <Insert> key toggles between inserting characters and replacing the existing ones. Start with this text:

```
/the fine pig
```

Move the cursor to the start of "fine" with <S-Left> twice (or <Left> eight times, if <S-Left> doesn't work). Now press <Insert> to switch to overstrike and type "great":

```
/the greatpig
```

Oops, we lost the space. Now, don't use <BS>, because it would delete the "t" (this is different from Replace mode). Instead, press <Insert> to switch from overstrike to inserting, and type the space:

```
/the great pig
```

Cancelling

You thought of executing a `:` or `/` command, but changed your mind. To get rid of what you already typed, without executing it, press CTRL-C or <Esc>.

Note: <Esc> is the universal "get out" key. Unfortunately, in the good old Vi pressing <Esc> in a command line executed the command! Since that might be considered to be a bug, Vim uses <Esc> to cancel the command. But with the `'cpoptions'` option it can be made Vi compatible. And when using a mapping (which might be written for Vi) <Esc> also works Vi compatible. Therefore, using CTRL-C is a method that always works.

If you are at the start of the command line, pressing <BS> will cancel the command. It's like deleting the "`:`" or "`/`" that the line starts with.

Command line abbreviations

Some of the "`:`" commands are really long. We already mentioned that "`:substitute`" can be abbreviated to "`:s`". This is a generic mechanism, all "`:`" commands can be abbreviated.

How short can a command get? There are 26 letters, and many more commands. For example, "`:set`" also starts with "`:s`", but "`:s`" doesn't start a "`:set`" command. Instead "`:set`" can be abbreviated to "`:se`".

When the shorter form of a command could be used for two commands, it stands for only one of them. There is no logic behind which one, you have to learn them. In the help files the shortest form that works is mentioned. For example:

```
:s[ubstitute]
```

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

In the user manual we will either use the full name of command, or a short version that is still readable. For example, "`:function`" can be abbreviated to "`:fu`". But since most people don't understand what that stands for, we will use "`:fun`". (Vim doesn't have a ":funny" command, otherwise ":fun" would be confusing too.)

It is recommended that in Vim scripts you write the full command name. That makes it easier to read back when you make later changes. Except for some often used commands like "`:w`" ("`:write`") and "`:r`" ("`:read`").

A particularly confusing one is ":end", which could stand for ":endif", ":endwhile" or ":endfunction". Therefore, always use the full name.

Short option names

In the user manual the long version of the option names is used. Many options also have a short name. Unlike ":" commands, there is only one short name that works. For example, the short name of 'autoindent' is 'ai'. Thus these two commands do the same thing:

```
:set autoindent
:set ai
```

You can find the full list of long and short names here: |:h option-list|.

Command line completion

This is one of those Vim features that, by itself, is a reason to switch from Vi to Vim. Once you have used this, you can't do without.

Suppose you have a directory that contains these files:

```
info.txt
intro.txt
bodyofthepaper.txt
```

To edit the last one, you use the command:

```
:edit bodyofthepaper.txt
```

It's easy to type this wrong. A much quicker way is:

```
:edit b<Tab>
```

Which will result in the same command. What happened? The <Tab> key does completion of the word before the cursor. In this case "b". Vim looks in the directory and finds only one file that starts with a "b". That must be the one you are looking for, thus Vim completes the file name for you.

Now type:

```
:edit i<Tab>
```

Vim will beep, and give you:

```
:edit info.txt
```

The beep means that Vim has found more than one match. It then uses the first match it found (alphabetically). If you press <Tab> again, you get:

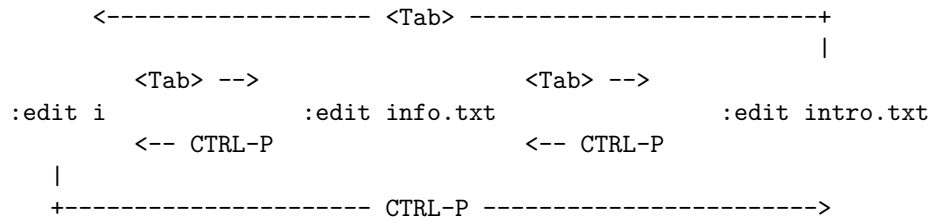
```
:edit intro.txt
```

Thus, if the first <Tab> doesn't give you the file you were looking for, press it again. If there are more matches, you will see them all, one at a time.

If you press <Tab> on the last matching entry, you will go back to what you first typed:

```
:edit i
```

Then it starts all over again. Thus Vim cycles through the list of matches. Use CTRL-P to go through the list in the other direction:



Context

When you type ":`set i`" instead of ":`edit i`" and press <Tab> you get:

```
:set icon
```

Hey, why didn't you get ":`set info.txt`"? That's because Vim has context sensitive completion. The kind of words Vim will look for depends on the command before it. Vim knows that you cannot use a file name just after a ":`set`" command, but you can use an option name.

Again, if you repeat typing the <Tab>, Vim will cycle through all matches. There are quite a few, it's better to type more characters first:

```
:set isk<Tab>
```

Gives:

```
:set iskeyword
```

Now type "=" and press <Tab>:

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

What happens here is that Vim inserts the old value of the option. Now you can edit it.

What is completed with <Tab> is what Vim expects in that place. Just try it out to see how it works. In some situations you will not get what you want. That's either because Vim doesn't know what you want, or because completion was not implemented for that situation. In that case you will get a <Tab> inserted (displayed as ^I).

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
```

Vim lists the matches and then comes back with the text you typed. You can now check the list for the item you wanted. If it isn't there, you can use <BS> to correct the word. If there are many matches, type a few more characters before pressing <Tab> to complete the rest.

If you have watched carefully, you will have noticed that "`incsearch`" doesn't start with "is". In this case "is" stands for the short name of "`incsearch`". (Many options have a short and a long name.) Vim is clever enough to know that you might have wanted to expand the short name of the option into the long name.

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`".

The `'wildmode'` option can be used to change the way completion works. The `'wildmenu'` option can be used to get a menu-like list of matches. Use the `'suffixes'` option to specify files that are less important and appear at the end of the list of files. The `'wildignore'` option specifies files that are not listed at all.

More about all of this here: |`:h cmdline-completion`|

Command line history

In chapter 3 we briefly mentioned the history. The basics are that you can use the <Up> key to recall an older command line. <Down> then takes you back to newer commands.

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. |`:h cmdline-history`|

Suppose you have done a ":`set`" command, typed ten more colon commands and then want to repeat that ":`set`" command again. You could press ":" and then ten times <Up>. There is a quicker way:

```
:se<Up>
```

Vim will now go back to the previous command that started with "`se`". You have a good chance that this is the ":`set`" command you were looking for. At least you should not have to press <Up> very often (unless ":`set`" commands is all you have done).

The <Up> key will use the text typed so far and compare it with the lines in the history. Only matching lines will be used.

If you do not find the line you were looking for, use <Down> to go back to what you typed and correct that. Or use CTRL-U to start all over again.

To see all the lines in the history:

```
:history
```

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

```
:history /
```

CTRL-P will work like <Up>, except that it doesn't matter what you already typed. Similarly for CTRL-N and <Down>. CTRL-P stands for previous, CTRL-N for next.

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:

```
+-------------------------------------+
|other window                         |
|~                                    |
|file.txt=============================|
|:e c                                 |
|:e config.h.in                       |
|:set path=.,/usr/include,,           |
|:set iskeyword=@,48-57,_,192-255     |
|:set is                              |
|:q                                   |
|:                                    |
|command-line=========================|
|                                     |
+-------------------------------------+
```

You are now in Normal mode. You can use the "`hjkl`" keys to move around. For example, move up with "`5k`" to the "`:e config.h.in`" line. Type "`$h`" to go to the "`i`" of "`in`" and type "`cwout`". Now you have changed the line to:

```
:e config.h.out
```

Now press <Enter> and this command will be executed. The command line window will close.

The <Enter> command will execute the line under the cursor. It doesn't matter whether Vim is in Insert mode or in Normal mode.

Changes in the command line window are lost. They do not result in the history to be changed. Except that the command you execute will be added to the end of the history, like with all executed commands.

The command line window is very useful when you want to have overview of the history, lookup a similar command, change it a bit and execute it. A search command can be used to find something.

In the previous example the "`?config`" search command could have been used to find the previous command that contains "config". It's a bit strange, because you are using a command line to search in the command line window. While typing that search command you can't open another command line window, there can be only one.

21. 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.

Suspend and resume

Like most Unix programs Vim can be suspended by pressing CTRL-Z. This stops Vim and takes you back to the shell it was started in. You can then do any other commands until you are bored with them. Then bring back Vim with the "`fg`" command.

CTRL-Z
{any sequence of shell commands}
fg

You are right back where you left Vim, nothing has changed.

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!

Only Unix has support for this. On other systems Vim will start a shell for you. This also has the functionality of being able to execute shell commands. But it's a new shell, not the one that you started Vim from.

When you are running the GUI you can't go back to the shell where Vim was started. CTRL-Z will minimize the Vim window instead.

Executing shell commands

To execute a single shell command from Vim use "`:!{command}`". For example, to see a directory listing:

```
:!ls
:!dir
```

The first one is for Unix, the second one for MS-Windows.

Vim will execute the program. When it ends you will get a prompt to hit <Enter>. This allows you to have a look at the output from the command before returning to the text you were editing.

The "`!`" is also used in other places where a program is run. Let's take a look at an overview:

```
:!{program}             execute {program}
:r !{program}              execute {program} and read its output
:w !{program}            execute {program} and send text to its input
:[range]!{program}  filter text through {program}
```

Notice that the presence of a range before "`!{program}`" makes a big difference. Without it executes the program normally, with the range a number of text lines is filtered through the program.

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.

When using the GUI the shell will be using the Vim window for its input and output. Since Vim is not a terminal emulator, this will not work perfectly. If you have trouble, try toggling the `'guipty'` option. If this still doesn't work well enough, start a new terminal to run the shell in. For example with:

```
:!xterm&
```

Remembering information; viminfo

After editing for a while you will have text in registers, marks in various files, a command line history filled with carefully crafted commands. When you exit Vim all of this is lost. But you can get it back!

The viminfo file is designed to store status information:

- Command-line and Search pattern history
- Text in registers
- Marks for various files
- The buffer list
- Global variables

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.

Take a look at how you can build up your own viminfo string. First, the `'` option is used to specify how many files for which you save marks (a-z). Pick a nice even number for this option (1000, for instance). Your command now looks like this:

```
:set viminfo='1000
```

The f option 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. You want this feature, so now you have this:

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

The `<` option controls how many lines are saved for each of the registers. By default, all the lines are saved. If 0, nothing is saved. To avoid adding thousands of lines to your viminfo file (which might never get used and makes starting Vim slower) you use a maximum of 500 lines:

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

Other options you might want to use:

`@`	number of lines to save from the input line history
`:`	number of lines to save from the command line history
`/`	number of lines to save from the search history
`r`	removable media, for which no marks will be stored (can be used several times)
`!`	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)

See the `'viminfo'` option and |`:h viminfo-file`| for more information.

When you run Vim multiple times, the last one exiting will store its information. This may cause information that previously exiting Vims stored to be lost. Each item can be remembered only once.

Getting back to where you stopped vim

You are halfway editing a file and it's time to leave for holidays. You exit Vim and go enjoy yourselves, forgetting all about your work. After a couple of weeks you start Vim, and type:

```
'0
```

And you are right back where you left Vim. So you can get on with your work.

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.

The |`:h :marks`| command is useful to find out where `'0` to `'9` will take you.

Getting back to some file

If you want to go back to a file that you edited recently, but not when exiting Vim, there is a slightly more complicated way. You can see a list of files by typing the command:

```
:oldfiles
   1: ~/.viminfo
   2: ~/text/resume.txt
   3: /tmp/draft
```

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

```
:e #<2
```

Instead of "`:e`" you can use any command that has a file name argument, 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
```

That `#<123` thing is a bit complicated when you just want to edit a file. Fortunately there is a simpler way:

```
:browse oldfiles
   1: ~/.viminfo
   2: ~/text/resume.txt
   3: /tmp/draft
   -- More --
```

You get the same list of files as with |`:h :oldfiles`|. If you want to edit "`resume.txt`" first press "`q`" to stop the listing. You will get a prompt:

```
Type number and <Enter> (empty cancels):
```

Type "2" and press <Enter> to edit the second file.

More info at |:h :oldfiles|, |:h v:oldfiles| and |:h c_#<|.

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
```

Obviously, the "w" stands for "write" and the "r" for "read".

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.

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 for session (actually, you can source any Vim script with -S, thus it might as well stand for "source").

The windows that were open are restored, with the same position and size as before. Mappings and option values are like before.

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
```

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
```

This first uses "`:wall`" to write all modified files. Then the current session is saved, using "`:mksession!`". This overwrites the previous session. The next time you load the secret session you can continue where you were at this point. And finally you load the new "boring" session.

If you open help windows, split and close various windows, and generally mess up the window layout, you can go back to the last saved session:

```
:source ~/.vim/boring.vim
```

Thus you have complete control over whether you want to continue next time where you are now, by saving the current setup in a session, or keep the session file as a starting point.

Another way of using sessions is to create a window layout that you like to use, and save this in a session. Then you can go back to this layout whenever you want.

For example, this is a nice layout to use:

```
+----------------------------------------+
|         VIM - main help file           |
|                                        |
|Move around:  Use the cursor keys, or "h|
|help.txt================================|
|explorer   |                            |
|dir        |~                           |
|dir        |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|~/=========|[No File]==================|
|                                        |
+----------------------------------------+
```

This has a help window at the top, so that you can read this text. The narrow vertical window on the left contains a file explorer. This is a Vim plugin that lists the contents of a directory. You can select files to edit there. More about this in the next chapter.

Create this from a just started Vim with:

```
:help
CTRL-W w
:vertical split ~/
```

You can resize the windows a bit to your liking. Then save the session with:

```
:mksession ~/.vim/mine.vim
```

Now you can start Vim with this layout:

```
vim -S ~/.vim/mine.vim
```

Hint: To open a file you see listed in the explorer window in the empty window, move the cursor to the filename and press "`O`". Double clicking with the mouse will also do this.

Unix and ms-windows

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 \.

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
```

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. For example, you have line numbers enabled with the `'number'` option and defined a few folds. Just like with sessions, you can remember this view on the file and restore it later. Actually, when you store a session, it stores the view of each window.

There are two basic ways to use views. The first is to let Vim pick a name for the view file. You can restore the view when you later edit the same file. To store the view for the current window:

```
:mkview
```

Vim will decide where to store the view. When you later edit the same file you get the view back with this command:

```
:loadview
```

That's easy, isn't it?

Now you want to view the file without the `'number'` option on, or with all folds open, you can set the options to make the window look that way. Then store this view with:

```
: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.

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
```

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: */
```

Put this line as one of the first or last five lines in the file. When editing the file, you will notice that `'shiftwidth'` will have been set to four. When editing another file, it's set back to the default value of eight.

For some files the modeline fits well in the header, thus it can be put at the top of the file. For text files and other files where the modeline gets in the way of the normal contents, put it at the end of the file.

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
```

The "`any-text`" indicates that you can put any text before and after the part that Vim will use. This allows making it look like a comment, like what was done above with /* and */.

The " `vim:`" part is what makes Vim recognize this line. There must be white space before "`vim`", or "`vim`" must be at the start of the line. Thus using something like "`gvim:`" will not work.

The part between the colons is a "`:set`" command. It works the same way as typing the "`:set`" command, except that you need to insert a backslash before a colon (otherwise it would be seen as the end of the modeline).

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.

For more details see |`:h modeline`|.

22. 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.

The file browser

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

```
:edit .
```

Through the magic of autocommands and Vim scripts, the window will be filled with the contents of the directory. It looks like this:

```
" ============================================================================
" Netrw Directory Listing                                        (netrw v109)
"   Sorted by      name
"   Sort sequence: [\/]$,\.h$,\.c$,\.cpp$,*,\.info$,\.swp$,\.o$\.obj$,\.bak$
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:exec
" ============================================================================
../
./
check/
Makefile
autocmd.txt
change.txt
eval.txt~
filetype.txt~
help.txt.info
```

You can see these items:

1. The name of the browsing tool and its version number
2. The name of the browsing directory
3. The method of sorting (may be by name, time, or size)
4. How names are to be sorted (directories first, then `*.h` files, `*.c` files, etc)
5. How to get help (use the <F1> key), and an abbreviated listing of available commands
6. A listing of files, including "`../`", which allows one to list the parent directory.

If you have syntax highlighting enabled, the different parts are highlighted so as to make it easier to spot them.

You can use Normal mode Vim commands to move around in the text. For example, move the cursor atop a file and press <Enter>; you will then be editing that file. To go back to the browser use "`:edit .`" again, or use "`:Explore`". CTRL-O also works.

Try using <Enter> while the cursor is atop a directory name. The result is that the file browser moves into that directory and displays the items found there. Pressing <Enter> on the first directory "`../`" moves you one level higher. Pressing "`-`" does the same thing, without the need to move to the "`../`" item first.

You can press <F1> to get help on the things you can do in the netrw file browser. This is what you get:

```
9. Directory Browsing     netrw-browse   netrw-dir   netrw-list   netrw-help

MAPS                                                                 netrw-maps
      <F1>.............Help.....................................|netrw-help|
      <cr>.............Browsing..................................|netrw-cr|
      <del>............Deleting Files or Directories..............|netrw-delete|
      -................Going Up..................................|netrw--|
      a................Hiding Files or Directories................|netrw-a|
      mb...............Bookmarking a Directory....................|netrw-mb|
      gb...............Changing to a Bookmarked Directory.........|netrw-gb|
      c................Make Browsing Directory The Current Dir....|netrw-c|
      d................Make A New Directory.......................|netrw-d|
      D................Deleting Files or Directories..............|netrw-D|
      <c-h>............Edit File/Directory Hiding List............|netrw-ctrl-h|
      i................Change Listing Style.......................|netrw-i|
      <c-l>............Refreshing the Listing.....................|netrw-ctrl-l|
      o................Browsing with a Horizontal Split...........|netrw-o|
      p................Use Preview Window.........................|netrw-p|
      P................Edit in Previous Window....................|netrw-p|
      q................Listing Bookmarks and History..............|netrw-q|
      r................Reversing Sorting Order....................|netrw-r|
    (etc)
```

The <F1> key thus brings you to a netrw directory browsing contents help page. It's a regular help page; use the usual |CTRL-]| to jump to tagged help items and |:h CTRL-O| to jump back.

To select files for display and editing: (with the cursor is atop a filename)

<enter>	Open the file in the current window.	|:h netrw-cr|
o	Horizontally split window and display file	|:h netrw-o|
v	Vertically split window and display file	|:h netrw-v|
p	Use the |:h preview-window|	|:h netrw-p|
P	Edit in the previous window	|:h netrw-P|
t	Open file in a new tab	|:h netrw-t|

The following normal-mode commands may be used to control the browser display:

- i Controls listing style (thin, long, wide, and tree). The long listing includes size and date information.
- s Repeatedly pressing s will change the way the files are sorted; one may sort on name, modification time, or size.
- r Reverse the sorting order.

As a sampling of extra normal-mode commands:

- c Change Vim's notion of the current directory to be the same as the browser directory. (see |:h g:netrw_keepdir! to control this, too)
- R Rename the file or directory under the cursor; a prompt will be issued for the new name.
- D Delete the file or directory under the cursor; a confirmation request will be issued.
- mb gb Make bookmark/goto bookmark

One may also use command mode; again, just a sampling:

`:Explore [directory]` Browse specified/current directory

`:NetrwSettings` A comprehensive list of your current netrw settings with help linkage.

The netrw browser is not limited to just your local machine; one may use urls such as: (that trailing `/` is important)

```
:Explore   ftp://somehost/path/to/dir/
:e scp://somehost/path/to/dir/
```

See |`:h netrw-browse`| for more.

The current directory

Just like the shell, Vim has the concept of a current directory. Suppose you are in your home directory and want to edit several files in a directory "`VeryLongFileName`". You could do:

```
:edit VeryLongFileName/file1.txt
:edit VeryLongFileName/file2.txt
:edit VeryLongFileName/file3.txt
```

To avoid much of the typing, do this:

```
:cd VeryLongFileName
:edit file1.txt
:edit file2.txt
:edit file3.txt
```

The "`:cd`" command changes the current directory. You can see what the current directory is with the "`:pwd`" command:

```
:pwd
/home/Bram/VeryLongFileName
```

Vim remembers the last directory that you used. Use "`cd -`" to go back to it. Example:

```
:pwd
/home/Bram/VeryLongFileName
:cd /etc
:pwd
/etc
:cd -
:pwd
/home/Bram/VeryLongFileName
:cd -
:pwd
/etc
```

Window local directory

When you split a window, both windows use the same current directory. When you want to edit a number of files somewhere else in the new window, you can make it use a different directory, without changing the current directory in the other window. This is called a local directory.

```
:pwd
/home/Bram/VeryLongFileName
:split
:lcd /etc
:pwd
/etc
CTRL-W w
:pwd
/home/Bram/VeryLongFileName
```

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.

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
```

Vim will find the file and edit it.

What if the file is not in the current directory? Vim will use the 'path' option to find the file. This option is a list of directory names where to look for your file.

Suppose you have your include files located in "c:/prog/include". This command will add it to the 'path' option:

```
:set path+=c:/prog/include
```

This directory is an absolute path. No matter where you are, it will be the same place. What if you have located files in a subdirectory, below where the file is? Then you can specify a relative path name. This starts with a dot:

```
:set path+=./proto
```

This tells Vim to look in the directory "proto", below the directory where the file in which you use "gf" is. Thus using "gf" on "inits.h" will make Vim look for "proto/inits.h", starting in the directory of the file.

Without the "./", thus "proto", Vim would look in the "proto" directory below the current directory. And the current directory might not be where the file that you are editing is located.

The 'path' option allows specifying the directories where to search for files in many more ways. See the help on the 'path' option.

The 'isfname' option is used to decide which characters are included in the file name, and which ones are not (e.g., the " character in the example above).

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"
```

This finds the file "stdio.h" in your value of 'path'. The quotes are necessary to have one argument |:h -+c|.

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.

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.

The argument of ":hide" is another command. ":hide" makes that command behave as if the 'hidden' option was set. You could also set this option yourself. The effect is that when any buffer is abandoned, it becomes hidden.

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

Inactive buffers

When a buffer has been used once, Vim remembers some information about it. When it is not displayed in a window and it is not hidden, it is still in the buffer list. This is called an inactive buffer. Overview:

Active	Appears in a window, text loaded.
Hidden	Not in a window, text loaded.
Inactive	Not in a window, no text loaded.

The inactive buffers are remembered, because Vim keeps information about them, like marks. And remembering the file name is useful too, so that you can see which files you have edited. And edit them again.

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 |`:h 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.

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
```

This works with a name as well.

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
```

Again, this also works with a name.

If you delete a buffer that was active (visible in a window), that window will be closed. If you delete the current buffer, the current window will be closed. If it was the last window, Vim will find another buffer to edit. You can't be editing nothing!

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.

23. 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.

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
```

You will notice the format in the message you get when editing a file. You don't see anything if you edit a native file format. Thus editing a Unix file on Unix won't result in a remark. But when you edit a dos file, Vim will notify you of this:

```
"/tmp/test" [dos] 3L, 71C
```

For a Mac file you would see "`[mac]`".

The detected file format is stored in the 'fileformat' option. To see which format you have, execute the following command:

```
:set fileformat?
```

The three names that Vim uses are:

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

Using the mac format

On Unix, <LF> is used to break a line. It's not unusual to have a <CR> character halfway a line. Incidentally, this happens quite often in Vi (and Vim) scripts.

On the Macintosh, where <CR> is the line break character, it's possible to have a <LF> character halfway a line.

The result is that it's not possible to be 100% sure whether a file containing both <CR> and <LF> characters is a Mac or a Unix file. Therefore, Vim assumes that on Unix you probably won't edit a Mac file, and doesn't check for this type of file. To check for this format anyway, add "`mac`" to `'fileformats'`:

```
:set fileformats+=mac
```

Then Vim will take a guess at the file format. Watch out for situations where Vim guesses wrong.

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.

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.

Files on the internet

Someone sends you an e-mail message, which refers to a file by its URL. For example:

```
    You can find the information here:
        ftp://ftp.vim.org/pub/vim/README
```

You could start a program to download the file, save it on your local disk and then start Vim to edit it.

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.

If something goes wrong you will get an error message. It's possible that the URL is wrong, you don't have permission to read it, the network connection is down, etc. Unfortunately, it's hard to tell the cause of the error. You might want to try the manual way of downloading the file.

Accessing files over the internet works with the netrw plugin. Currently URLs with these formats are recognized:

```
    ftp://         uses ftp
    rcp://      uses rcp
    scp://      uses scp
    http://         uses wget reading only
```

Vim doesn't do the communication itself, it relies on the mentioned programs to be available on your computer. On most Unix systems "`ftp`" and "`rcp`" will be present. "`scp`" and "`wget`" might need to be installed.

Vim detects these URLs for each command that starts editing a new file, also with "`:edit`" and "`:split`", for example. Write commands also work, except for `http://`.

For more information, also about passwords, see |`:h netrw`|.

Encryption

Some information you prefer to keep to yourself. For example, when writing a test on a computer that students also use. You don't want clever students to figure out a way to read the questions before the exam starts. Vim can encrypt the file for you, which gives you some protection.

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:
```

Carefully type the secret key now. You cannot see the characters you type, they will be replaced by stars. To avoid the situation that a typing mistake will cause trouble, Vim asks you to enter the key again:

```
Enter same key again:
```

You can now edit this file normally and put in all your secrets. When you finish editing the file and tell Vim to exit, the file is encrypted and written.

When you edit the file with Vim, it will ask you to enter the same key again. You don't need to use the "`-x`" argument. You can also use the normal "`:edit`" command. 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).

Switching encryption on and off

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

```
:set key=
```

The next time you write the file this will be done without encryption.

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: ******
```

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
```

Since there is no swapfile, recovery will be impossible. Save the file a bit more often to avoid the risk of losing your changes.

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.

If you really want to secure the contents of a file, edit it only on a portable computer not connected to a network, use good encryption tools, and keep the computer locked up in a big safe when not in use.

Binary files

You can edit binary files with Vim. Vim wasn't really made for this, thus there are a few restrictions. But you can read a file, change a character and write it back, with the result that only that one character was changed and the file is identical otherwise.

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
```

This sets the `'binary'` option. The effect of this is that unexpected side effects are turned off. For example, `'textwidth'` is set to zero, to avoid automatic formatting of lines. And files are always read in Unix file format.

Binary mode can be used to change a message in a program. Be careful not to insert or delete any characters, it would stop the program from working. Use "`R`" to enter replace mode.

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. The output, when the cursor is on an <Esc>, looks like this:

```
    <^[>  27,  Hex 1b,  Octal 033
```

There might not be many line breaks in the file. To get some overview switch the `'wrap'` option off:

```
:set nowrap
```

Byte position

To see on which byte you are in the file use this command:

```
g CTRL-G
```

The output is verbose:

```
Col 9-16 of 9-16; Line 277 of 330; Word 1806 of 2058; Byte 10580 of 12206
```

The last two numbers are the byte position in the file and the total number of bytes. This takes into account how `'fileformat'` changes the number of bytes that a line break uses. To move to a specific byte in the file, use the "`go`" command. For example, to move to byte 2345:

```
2345go
```

Using xxd

A real binary editor shows the text in two ways: as it is and in hex format. You can do this in Vim by first converting the file with the "`xxd`" program. This comes with Vim.

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.

See the manual page of xxd for more information.

Compressed files

This is easy: You can edit a compressed file just like any other file. The "`gzip`" plugin takes care of decompressing the file when you edit it. And compressing it again when you write it.

These compression methods are currently supported:

- .Z compress
- .gz gzip
- .bz2 bzip2

Vim uses the mentioned programs to do the actual compression and decompression. You might need to install the programs first.

24. 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.

Making corrections

The <BS> key was already mentioned. It deletes the character just before the cursor. The <Del> key does the same for the character under (after) the cursor.

```
When you typed a whole word wrong, use CTRL-W:

    The horse had fallen to the sky
                                  CTRL-W
    The horse had fallen to the
```

If you really messed up a line and want to start over, use CTRL-U to delete it. This keeps the text after the cursor and the indent. Only the text from the first non-blank to the cursor is deleted. With the cursor on the "`f`" of "`fallen`" in the next line pressing CTRL-U does this:

```
    The horse had fallen to the
                 CTRL-U
    fallen to the
```

When you spot a mistake a few words back, you need to move the cursor there to correct it. For example, you typed this:

```
    The horse had follen to the ground
```

You need to change "`follen`" to "`fallen`". With the cursor at the end, you would type this to correct it:

```
                              <Esc>4blraA

    get out of Insert mode    <Esc>
    four words back                4b
    move on top of the "o"           l
    replace with "a"                  ra
    restart Insert mode                 A
```

Another way to do this:

```
     <C-Left><C-Left><C-Left><C-Left><Right><Del>a<End>

    four words back          <C-Left><C-Left><C-Left><C-Left>
    move on top of the "o"             <Right>
    delete the "o"                        <Del>
    insert an "a"                              a
    go to end of the line                       <End>
```

This uses special keys to move around, while remaining in Insert mode. This resembles what you would do in a modeless editor. It's easier to remember, but takes more time (you have to move your hand from the letters to the cursor keys, and the <End> key is hard to press without looking at the keyboard). These

special keys are most useful when writing a mapping that doesn't leave Insert mode. The extra typing doesn't matter then. An overview of the keys you can use in Insert mode:

<C-Home>	to start of the file
<PageUp>	a whole screenful up
<Home>	to start of line
<S-Left>	one word left
<C-Left>	one word left
<S-Right>	one word right
<C-Right>	one word right
<End>	to end of the line
<PageDown>	a whole screenful down
<C-End>	to end of the file

There are a few more, see |`:h ins-special-special`|.

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
```

When you now type a text like "`(example)`", as soon as you type the `)` Vim will briefly move the cursor to the matching `(`, keep it there for half a second, and move back to where you were typing. In case there is no matching `(`, Vim will beep. Then you know that you might have forgotten the `(` somewhere, or typed a `)` too many. The match will also be shown for `[]` and `{}` pairs. You don't have to wait with typing the next character, as soon as Vim sees it the cursor will move back and inserting continues as before. 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.

Completion

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

Suppose, for example, that you are creating a C program and want to type in the following:

```
total = ch_array[0] + ch_array[1] + ch_array[2];
```

You start by entering the following:

```
total = ch_array[0] + ch_
```

At this point, you tell Vim to complete the word using the command CTRL-P. Vim searches for a word that starts with what's in front of the cursor. In this case, it is "`ch_`", which matches with the word `ch_array`. So typing CTRL-P gives you the following:

```
total = ch_array[0] + ch_array
```

After a little more typing, you get this (ending in a space):

```
total = ch_array[0] + ch_array[1] +
```

If you now type CTRL-P Vim will search again for a word that completes the word before the cursor. Since there is nothing in front of the cursor, it finds the first word backwards, which is "`ch_array`". Typing CTRL-P again gives you the next word that matches, in this case "`total`". A third CTRL-P searches further back. If there is nothing else, it causes the editor to run out of words, so it returns to the original text, which is nothing. A fourth CTRL-P causes the editor to start over again with "`ch_array`".

To search forward, use CTRL-N. Since the search wraps around the end of the file, CTRL-N and CTRL-P will find the same matches, but in a different sequence. Hint: CTRL-N is Next-match and CTRL-P is Previous-match.

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

Options

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

After each of them CTRL-N can be used to find the next match, CTRL-P to find the previous match.

More information for each of these commands here: |`:h ins-completion`|.

Completing file names

Let's take CTRL-X CTRL-F as an example. This will find file names. It scans the current directory for files and displays each one that matches the word in front of the cursor.

Suppose, for example, that you have the following files in the current directory:

```
main.c  sub_count.c  sub_done.c  sub_exit.c
```

Now enter Insert mode and start typing:

```
The exit code is in the file sub
```

At this point, you enter the command CTRL-X CTRL-F. Vim now completes the current word "`sub`" by looking at the files in the current directory. The first match is `sub_count.c`. This is not the one you want, so you match the next file by typing CTRL-N. This match is `sub_done.c`. Typing CTRL-N again takes you to `sub_exit.c`. The results:

```
The exit code is in the file sub_exit.c
```

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):

```
the file is found in /usr/
```

If you now press CTRL-N you go back to "`/u`". Instead, to accept the "`/usr/`" and go one directory level deeper, use CTRL-X CTRL-F again:

```
the file is found in /usr/X11R6/
```

The results depend on what is found in your file system, of course. The matches are sorted alphabetically.

Completing in source code

Source code files are well structured. That makes it possible to do completion in an intelligent way. In Vim this is called Omni completion. In some other editors it's called intellisense, but that is a trademark.

The key to Omni completion is CTRL-X CTRL-O. Obviously the O stands for Omni here, so that you can remember it easier. Let's use an example for editing C source:

```
{
    struct foo *p;
    p->
```

The cursor is after "`p->`". Now type CTRL-X CTRL-O. Vim will offer you a list of alternatives, which are the items that "`struct foo`" contains. That is quite different from using CTRL-P, which would complete any word, while only members of "`struct foo`" are valid here.

For Omni completion to work you may need to do some setup. At least make sure filetype plugins are enabled. Your vimrc file should contain a line like this:

```
filetype plugin on
```

Or:

```
filetype plugin indent on
```

For C code you need to create a tags file and set the `'tags'` option. That is explained |`:h ft-c-omni`|. For other filetypes you may need to do something similar, look below |`:h compl-omni-filetypes`|. It only works for specific filetypes. Check the value of the `'omnifunc'` option to find out if it would work.

Repeating an insert

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

Assume, for example, that you have a file that begins with the following:

```
"file.h"
/* Main program begins */
```

You edit this file by inserting "`#include `" at the beginning of the first line:

```
#include "file.h"
/* Main program begins */
```

You go down to the beginning of the next line using the commands "`j^`". You now start to insert a new "`#include`" line. So you type:

```
i CTRL-A
```

The result is as follows:

```
#include "file.h"
#include /* Main program begins */
```

The "`#include `" was inserted because CTRL-A inserts the text of the previous insert. Now you type "`main.h`"<Enter> to finish the line:

```
#include "file.h"
#include "main.h"
/* Main program begins */
```

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

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.

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
```

`"v` is the register specification, "`yiw`" is yank-inner-word. Now edit the file where the new line is to be inserted, and type the first letters:

```
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}`.

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. This is triggered by typing a character that can't be part of a word, for example a space:

What Is Entered	What You See
I saw the a	I saw the a
I saw the ad	I saw the ad
I saw the ad<Space>	I saw the advertisement<Space>

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. The comment starts with `#b`, which draws the top line. I then type the comment text and use `#e` to draw the bottom line.

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

It's very common to make the same typing mistake every time. For example, typing "teh" instead of "the". You can fix this with an abbreviation:

```
:abbreviate teh the
```

You can add a whole list of these. Add one each time you discover a common mistake.

Listing abbreviations

The "`:abbreviate`" command lists the 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
```

While you type this, you will notice that @f is expanded to "fresh". Don't worry about this, Vim understands it anyway (except when you have an abbreviation for "fresh", but that's very unlikely).

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.

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>
```

Inserts an escape character. Thus you don't leave Insert mode. (Don't type the space after CTRL-V, it's only to make this easier to read).

Note: On MS-Windows CTRL-V is used to paste text. Use CTRL-Q instead of CTRL-V. On Unix, on the other hand, CTRL-Q does not work on some terminals, because it has a special meaning.

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
```

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 copyright, for example, you press three keys:

```
CTRL-K Co
```

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

```
:digraphs
```

Vim will display the digraph table. Here are three lines of it:

```
AC ~_ 159  NS |  160  !I ¡  161  Ct ¢  162  Pd £  163  Cu ¤  164  Ye ¥  165
BB ¦  166  SE §  167  ': ¨  168  Co ©  169  -a ª  170  << «  171  NO ¬  172
--    173  Rg ®  174  'm ¯  175  DG °  176  +- ±  177  2S ²  178  3S ³  179
```

This shows, for example, that the digraph you get by typing CTRL-K Pd is the character (£). This is character number 163 (decimal).

Pd is short for Pound. Most digraphs are selected to give you a hint about the character they will produce. If you look through the list you will understand the logic.

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.

Note: The digraphs depend on the character set that Vim assumes you are using. On MS-DOS they are different from MS-Windows. Always use ":digraphs" to find out which digraphs are currently available.

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
```

More information about digraphs here: |:h digraphs|

Another way to insert special characters is with a keymap. More about that here: |Entering language text|

Normal mode commands

Insert mode offers a limited number of commands. In Normal mode you have many more. When you want to use one, you usually leave Insert mode with <Esc>, execute the Normal mode command, and re-enter Insert mode with "i" or "a".

There is a quicker way. 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`.

25. 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.

Breaking lines

Vim has a number of functions that make dealing with text easier. By default, the editor does not perform automatic line breaks. In other words, you have to press <Enter> yourself. This is useful when you are writing programs where you want to decide where the line ends. It is not so good when you are creating documentation and want the text to be at most 70 character wide.

If you set the `'textwidth'` option, Vim automatically inserts line breaks. Suppose, for example, that you want a very narrow column of only 30 characters. You need to execute the following command:

```
:set textwidth=30
```

Now you start typing (ruler added):

```
          1         2         3
123456789012345678901234567890123456
I taught programming for a whi
```

If you type "`l`" next, this makes the line longer than the 30-character limit. When Vim sees this, it inserts a line break and you get the following:

```
          1         2         3
123456789012345678901234567890123456
I taught programming for a
whil
```

Continuing on, you can type in the rest of the paragraph:

```
          1         2         3
123456789012345678901234567890123456
I taught programming for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

You do not have to type newlines; Vim puts them in automatically.

Note: The `'wrap'` option makes Vim display lines with a line break, but this doesn't insert a line break in the file.

Reformatting

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.
```

This does not look good. To get the paragraph into shape you use the "`gq`" operator.

Let's first use this with a Visual selection. Starting from the first line, type:

```
v4jgq
```

"`v`" to start Visual mode, "`4j`" to move to the end of the paragraph and then the "`gq`" operator. The result is:

```
          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.
```

Note: there is a way to do automatic formatting for specific types of text layouts, see |`:h auto-format`|.

Since "`gq`" is an operator, you can use one of the three ways to select the text it works on: With Visual mode, with a movement and with a text object.

The example above could also be done with "`gq4j`". That's less typing, but you have to know the line count. A more useful motion command is "`}`". This moves to the end of a paragraph. Thus "`gq}`" formats from the cursor to the end of the current paragraph.

A very useful text object to use with "`gq`" is the paragraph. Try this:

```
gqap
```

"`ap`" stands for "a-paragraph". This formats the text of one paragraph (separated by empty lines). Also the part before the cursor.

If you have your paragraphs separated by empty lines, you can format the whole file by typing this:

```
gggqG
```

"`gg`" to move to the first line, "`gqG`" to format until the last line.

Warning: If your paragraphs are not properly separated, they will be joined together. A common mistake is to have a line with a space or tab. That's a blank line, but not an empty line.

Vim is able to format more than just plain text. See |`:h fo-table`| for how to change this. See the `'joinspaces'` option to change the number of spaces used after a full stop.

It is possible to use an external program for formatting. This is useful if your text can't be properly formatted with Vim's builtin command. See the `'formatprg'` option.

Aligning text

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

Similarly, the ":right" command right-justifies the text:

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

gives this result:

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

Left alignment

Finally there is this command:

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

Unlike ":center" and ":right", however, the argument to ":left" is not the length of the line. Instead it is the left margin. If it is omitted, the text will be put against the left side of the screen (using a zero margin would do the same). If it is 5, the text will be indented 5 spaces. For example, use these commands:

```
:1left 5
:2,5left
```

This 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.
```

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
```

Indents and tabs

Indents can be used to make text stand out from the rest. The example texts in this manual, for example, are indented by eight spaces or a tab. You would normally enter this by typing a tab at the start of each line. Take this text:

```
    the first line
    the second line
```

This is entered by typing a tab, some text, <Enter>, tab and more text.

The `'autoindent'` option inserts indents automatically:

```
:set autoindent
```

When a new line is started it gets the same indent as the previous line. In the above example, the tab after the <Enter> is not needed anymore.

Increasing indent

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

The amount of indent added is specified with the `'shiftwidth'` option. The default value is 8. To make "`>>`" insert four spaces worth of indent, for example, type this:

```
:set shiftwidth=4
```

When used on the second line of the example text, this is what you get:

```
    the first line
        the second line
```

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

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.

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.

Dealing with long lines

Sometimes you will be editing a file that is wider than the number of columns in the window. When that occurs, Vim wraps the lines so that everything fits on the screen.

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.

When you move the cursor to a character that can't be seen, Vim will scroll the text to show it. This is like moving a viewport over the text in the horizontal direction.

By default, Vim does not display a horizontal scrollbar in the GUI. If you want to enable one, use the following command:

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

Let's attempt to show this with one line of text. The cursor is on the "w" of "which". The "current window" above the line indicates the text that is currently visible. The "window"s below the text indicate the text that is visible after the command left of it.

```
                    |<-- current window -->|
    some long text, part of which is visible in the window
ze    |<--      window    -->|
zH     |<--      window    -->|
4zh           |<--      window    -->|
zh               |<--     window     -->|
zl                 |<--  window        -->|
4zl                   |<--      window    -->|
zL                          |<--     window     -->|
zs                         |<--   window       -->|
```

Moving with wrap off

When 'wrap' is off and the text has scrolled horizontally, you can use the following commands to move the cursor to a character you can see. Thus text left and right of the window is ignored. These never cause the text to scroll:

g0	to first visible character in this line
g^	to first non-blank visible character in this line
gm	to middle of this line
g$	to last visible character in this line

```
        |<--     window    -->|
some long    text, part of which is visible
         g0  g^    gm         g$
```

Breaking at words

When preparing text for use by another program, you might have to make paragraphs without a line break. A disadvantage of using 'nowrap' is that you can't see the whole sentence you are working on. When 'wrap' is on, words are broken halfway, which makes them hard to read.

A good solution for editing this kind of paragraph is setting the 'linebreak' option. Vim then breaks lines at an appropriate place when displaying the line. The text in the file remains unchanged.

Without 'linebreak' text might look like this:

```
+---------------------------------+
|letter generation program for a b|
|ank.  They wanted to send out a s|
|pecial, personalized letter to th|
|eir richest 1000 customers.  Unfo|
|rtunately for the programmer, he |
+---------------------------------+
```

After:

```
:set linebreak
```

it looks like this:

```
+---------------------------------+
|letter generation program for a  |
|bank.  They wanted to send out a |
|special, personalized letter to  |
|their richest 1000 customers.    |
|Unfortunately for the programmer,|
+---------------------------------+
```

Related options:

- `'breakat'` specifies the characters where a break can be inserted.
- `'showbreak'` specifies a string to show at the start of broken line.

Set `'textwidth'` to zero to avoid a paragraph to be split.

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
```

Turning a paragraph into one line

If you want to import text into a program like MS-Word, each paragraph should be a single line. If your paragraphs are currently separated with empty lines, this is how you turn each paragraph into a single line:

```
:g/./,/^$/join
```

That looks complicated. Let's break it up in pieces:

```
:g/./          A ":global" command that finds all lines that contain at least one character.
     ,/^$/     A range, starting from the current line (the non-empty line) until an empty line.
          join The ":join" command joins the range of lines together into one line.
```

Starting with this text, containing eight lines broken at column 30:

```
+----------------------------------+
|A letter generation program       |
|for a bank.  They wanted to       |
|send out a special,               |
|personalized letter.              |
|                                  |
|To their richest 1000             |
|customers.  Unfortunately for     |
|the programmer,                   |
+----------------------------------+
```

You end up with two lines:

```
+----------------------------------+
|A letter generation program for a |
|bank.  They wanted to send out a s|
|pecial, personalized letter.      |
|To their richest 1000 customers.  |
|Unfortunately for the programmer, |
+----------------------------------+
```

Note that this doesn't work when the separating line is blank but not empty; when it contains spaces and/or tabs. This command does work with blank lines:

```
:g/\S/,/^\s*$/join
```

This still requires a blank or empty line at the end of the file for the last paragraph to be joined.

Editing tables

Suppose you are editing a table with four columns:

```
nice table    test 1    test 2      test 3
input A       0.534
input B       0.913
```

You need to enter numbers in the third column. You could move to the second line, use "`A`", enter a lot of spaces and type the text.

For this kind of editing there is a special option:

```
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.

Move the cursor by searching for the header of the last column:

```
/test 3
```

Now press "`j`" and you are right where you can enter the value for "`input A`". Typing "`0.693`" results in:

```
nice table    test 1     test 2  test 3
input A       0.534              0.693
input B       0.913
```

Vim has automatically filled the gap in front of the new text for you. Now, to enter the next field in this column use "`Bj`". "`B`" moves back to the start of a white space separated word. Then "`j`" moves to the place where the next field can be entered.

Note: You can move the cursor anywhere in the display, also beyond the end of a line. But Vim will not insert spaces there, until you insert a character in that position.

Copying a column

You want to add a column, which should be a copy of the third column and placed before the "test 1" column. Do this in seven steps:

1. Move the cursor to the left upper corner of this column, e.g., with "`/test 3`".
2. Press CTRL-V to start blockwise Visual mode.

3. Move the cursor down two lines with "2j". You are now in "virtual space": the "input B" line of the "test 3" column.
4. Move the cursor right, to include the whole column in the selection, plus the space that you want between the columns. "9l" should do it.
5. Yank the selected rectangle with "y".
6. Move the cursor to "test 1", where the new column must be placed.
7. Press "P".

The result should be:

```
nice table    test 3    test 1     test 2      test 3
input A       0.693     0.534                  0.693
input B                 0.913
```

Notice that the whole "test 1" column was shifted right, also the line where the "test 3" column didn't have text.

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.

Suppose you have a line in a table that contains both tabs and other characters. Use "rx" on the first tab:

```
inp     0.693   0.534   0.693

        |
   rx   |
        V

inpx0.693   0.534       0.693
```

The layout is messed up. To avoid that, use the "gr" command:

```
inp     0.693   0.534   0.693

        |
  grx   |
        V

inpx    0.693   0.534   0.693
```

What happens is that the "gr" command makes sure the new character takes the right amount of screen space. Extra spaces or tabs are inserted to fill the gap. Thus what actually happens is that a tab is replaced by "x" and then blanks added to make the text after it keep its place. In this case a tab is inserted.

When you need to replace more than one character, you use the "R" command to go to Replace mode (see |Replace mode|). This messes up the layout and replaces the wrong characters:

```
    inp 0       0.534   0.693

        |
 R0.786 |
        V

    inp 0.78634 0.693
```

The "gR" command uses Virtual Replace mode. This preserves the layout:

```
    inp     0       0.534   0.693

            |
    gR0.786 |
            V

    inp     0.786   0.534   0.693
```

26. 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.

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.

Suppose you have some lines where you want to change "2001" to "2002" and "2000" to "2001":

```
The financial results for 2001 are better
than for 2000.  The income increased by 50%,
even though 2001 had more rain than 2000.
            2000        2001
income      45,403      66,234
```

First change "2001" to "2002". Select the lines in Visual mode, and use:

```
:s/2001/2002/g
```

Now use "gv" to reselect the same text. It doesn't matter where the cursor is. Then use ":s/2000/2001/g" to make the second change.

Obviously, you can repeat these changes several times.

Add and subtract

When repeating the change of one number into another, you often have a fixed offset. In the example above, one was added to each year. Instead of typing a substitute command for each year that appears, the CTRL-A command can be used.

Using the same text as above, search for a year:

```
/19[0-9][0-9]\|20[0-9][0-9]
```

Now press CTRL-A. The year will be increased by one:

```
The financial results for 2002 are better
than for 2000.  The income increased by 50%,
even though 2001 had more rain than 2000.
            2000        2001
income      45,403      66,234
```

Use "n" to find the next year, and press "." to repeat the CTRL-A ("." is a bit quicker to type). Repeat "n" and "." for all years that appear.

Hint: set the 'hlsearch' option to see the matches you are going to change, then you can look ahead and do it faster.

Adding more than one can be done by prepending the number to CTRL-A. Suppose you have this list:

```
1.  item four
2.  item five
3.  item six
```

Move the cursor to "1." and type:

```
3 CTRL-A
```

The "1." will change to "4.". Again, you can use "`.`" to repeat this on the other numbers.

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
```

The CTRL-X command does subtraction in a similar way.

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`").

Using Vim from a shell script

Suppose you have a lot of files in which you need to change the string "-person-" to "Jones" and then print it. How do you do that? One way is to do a lot of typing. The other is to write a shell script to do the work.

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 for-done loop is a shell construct to repeat the two lines in between, while the `$file` variable is set to a different file name each time.

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).

Reading from stdin

Vim can read text on standard input. Since the normal way is to read commands there, you must tell Vim to read text instead. This is done by passing the "`-`" argument in place of a file. Example:

```
ls | vim -
```

This allows you to edit the output of the "`ls`" command, without first saving the text in a file.

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 -
```

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.

27. Search commands and patterns

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

Ignoring case

By default, Vim's searches are case sensitive. Therefore, "include", "INCLUDE", and "Include" are three different words and a search will match only one of them.

Now switch on the `'ignorecase'` option:

```
:set ignorecase
```

Search for "include" again, and now it will match "Include", "INCLUDE" and "InClUDe". (Set the `'hlsearch'` option to quickly see where a pattern matches.)

You can switch this off again with:

```
:set noignorecase
```

But let's keep it set, and search for "INCLUDE". It will match exactly the same text as "include" did. Now set the `'smartcase'` option:

```
:set ignorecase smartcase
```

If you have a pattern with at least one uppercase character, the search becomes case sensitive. The idea is that you didn't have to type that uppercase character, so you must have done it because you wanted case to match. That's smart!

With these two options set you find the following matches:

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

Case in one pattern

If you want to ignore case for one specific pattern, you can do this by prepending the "`\c`" string. Using "`\C`" will make the pattern to match case. This overrules the `'ignorecase'` and `'smartcase'` options, when "`\c`" or "`\C`" is used their value doesn't matter.

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 `'magic'` 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.

Note: If your search takes much longer than you expected, you can interrupt it with CTRL-C on Unix and CTRL-Break on MS-DOS and MS-Windows.

Wrapping around the file end

By default, a forward search starts searching for the given string at the current cursor location. It then proceeds to the end of the file. If it has not found the string by that time, it starts from the beginning and searches from the start of the file to the cursor location.

Keep in mind that when repeating the "`n`" command to search for the next match, you eventually get back to the first match. If you don't notice this you keep searching forever! To give you a hint, Vim displays this message:

```
search hit BOTTOM, continuing at TOP
```

If you use the "`?`" command, to search in the other direction, you get this message:

```
search hit TOP, continuing at BOTTOM
```

Still, you don't know when you are back at the first match. One way to see this is by switching on the `'ruler'` option:

```
:set ruler
```

Vim will display the cursor position in the lower righthand corner of the window (in the status line if there is one). It looks like this:

```
101,29        84%
```

The first number is the line number of the cursor. Remember the line number where you started, so that you can check if you passed this position again.

Not wrapping

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
```

Thus you can find all matches by going to the start of the file with "`gg`" and keep searching until you see this message.

If you search in the other direction, using "`?`", you get:

```
E384: search hit TOP without match for: forever
```

Offsets

By default, the search command leaves the cursor positioned on the beginning of the pattern. You can tell Vim to leave it some other place by specifying an offset. For the forward search command "`/`", the offset is specified by appending a slash (`/`) and the offset:

```
/default/2
```

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.

Character offsets

The "`e`" offset indicates an offset from the end of the match. It moves the cursor onto the last character of the match. The command:

```
/const/e
```

puts the cursor on the "`t`" of "`const`".

From that position, adding a number moves forward that many characters. This command moves to the character just after the match:

```
/const/e+1
```

A positive number moves the cursor to the right, a negative number moves it to the left. For example:

```
/const/e-1
```

moves the cursor to the "`s`" of "`const`".

If the offset begins with "`b`", the cursor moves to the beginning of the pattern. That's not very useful, since leaving out the "`b`" does the same thing. It does get useful when a number is added or subtracted. The cursor then goes forward or backward that many characters. For example:

```
/const/b+2
```

Moves the cursor to the beginning of the match and then two characters to the right. Thus it lands on the "`n`".

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:

```
//
```

Searching backwards

The "?" command uses offsets in the same way, but you must use "?" to separate the offset from the pattern, instead of "/":

```
?const?e-2
```

The "b" and "e" keep their meaning, they don't change direction with the use of "?".

Start position

When starting a search, it normally starts at the cursor position. When you specify a line offset, this can cause trouble. For example:

```
/const/-2
```

This finds the next word "const" and then moves two lines up. If you use "n" to search again, Vim could start at the current position and find the same "const" match. Then using the offset again, you would be back where you started. You would be stuck!

It could be worse: Suppose there is another match with "const" in the next line. Then repeating the forward search would find this match and move two lines up. Thus you would actually move the cursor back!

When you specify a character offset, Vim will compensate for this. Thus the search starts a few characters forward or backward, so that the same match isn't found again.

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".

Specific counts

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 |`:h 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 remove both 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.

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.

Character ranges

To match "a", "b" or "c" you could use "`/a\|b\|c`". When you want to match all letters from "a" to "z" this gets very long. There is a shorter method:

```
/[a-z]
```

The `[]` construct matches a single character. Inside you specify which characters to match. You can include a list of characters, like this:

```
/[0123456789abcdef]
```

This will match any of the characters included. For consecutive characters you can specify the range. "`0-3`" stands for "`0123`". "`w-z`" stands for "`wxyz`". Thus the same command as above can be shortened to:

```
/[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 |`:h /[]`| 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 |`:h /\s`| for the whole list of these ranges.

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	

Matching a line break

Vim can find a pattern that includes a line break. You need to specify where the line break happens, because all items mentioned so far don't match 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.

Examples

Here are a few search patterns you might find useful. This shows how the items mentioned above can be combined.

Finding a california license plate

A sample license plate number is "1MGU103". It has one digit, three uppercase letters and three digits. Directly putting this into a search pattern:

```
/\d\u\u\u\d\d\d
```

Another way is to specify that there are three digits and letters with a count:

```
/\d\u\{3}\d\{3}
```

Using `[]` ranges instead:

```
/[0-9][A-Z]\{3}[0-9]\{3}
```

Which one of these you should use? Whichever one you can remember. The simple way you can remember is much faster than the fancy way that you can't. If you can remember them all, then avoid the last one, because it's both more typing and slower to execute.

Finding an identifier

In C programs (and many other computer languages) an identifier starts with a letter and further consists of letters and digits. Underscores can be used too. This can be found with:

```
/\<\h\w*\>
```

"`\<`" and "`\>`" are used to find only whole words. "`\h`" stands for "`[A-Za-z_]`" and "`\w`" for "`[0-9A-Za-z_]`".

Note: "`\<`" and "`\>`" depend on the `'iskeyword'` option. If it includes "`-`", for example, then "ident-" is not matched. In this situation use:

```
/\w\@<!\h\w*\w\@!
```

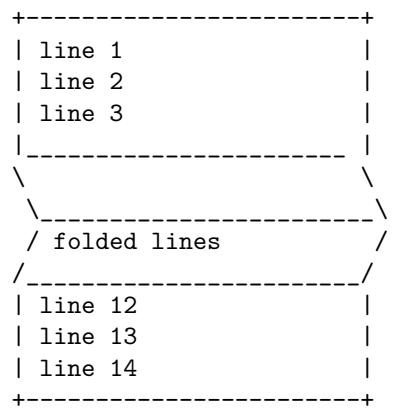
This checks if "`\w`" does not match before or after the identifier. See |`:h /\@<!`| and |`:h /\@!`|.

28. 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.

What is folding?

Folding is used to show a range of lines in the buffer as a single line on the screen. Like a piece of paper which is folded to make it shorter:



The text is still in the buffer, unchanged. Only the way lines are displayed is affected by folding.

The advantage of folding is that you can get a better overview of the structure of text, by folding lines of a section and replacing it with a line that indicates that there is a section.

Manual folding

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

```
zfap
```

You will see that the paragraph is replaced by a highlighted line. You have created a fold. |`:h zf`| is an operator and |`:h ap`| a text object selection. You can use the |`:h zf`| operator with any movement command to create a fold for the text that it moved over. |`:h 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
```

All the folding commands start with "`z`". With some fantasy, this looks like a folded piece of paper, seen from the side. The letter after the "`z`" has a mnemonic meaning to make it easier to remember the commands:

zf	F-old creation
zo	O-pen a fold
zc	C-lose a fold

Folds can be nested: A region of text that contains folds can be folded again. For example, you can fold each paragraph in this section, and then fold all the sections in this chapter. Try it out. You will notice that opening the fold for the whole chapter will restore the nested folds as they were, some may be open and some may be closed.

Suppose you have created several folds, and now want to view all the text. You could go to each fold and type "`zo`". To do this faster, use this command:

```
zr
```

This will R-educe the folding. The opposite is:

```
zm
```

This folds M-ore. You can repeat "`zr`" and "`zm`" to open and close nested folds of several levels.

If you have nested several levels deep, you can open all of them with:

```
zR
```

This R-educes folds until there are none left. And you can close all folds with:

```
zM
```

This folds M-ore and M-ore.

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

- create folds to get overview on your file
- move around to where you want to do your work
- do |`:h zi`| to look at the text and edit it
- do |`:h zi`| again to go back to moving around

More about manual folding in the reference manual: |`:h fold-manual`|

Working with folds

When some folds are closed, movement commands like "`j`" and "`k`" move over a fold like it was a single, empty line. This allows you to quickly move around over folded text.

You can yank, delete and put folds as if it was a single line. This is very useful if you want to reorder functions in a program. First make sure that each fold contains a whole function (or a bit less) by selecting the right `'foldmethod'`. Then delete the function with "`dd`", move the cursor and put it with "`p`". If some lines of the function are above or below the fold, you can use Visual selection:

- put the cursor on the first line to be moved
- hit "`V`" to start Visual mode
- put the cursor on the last line to be moved
- hit "`d`" to delete the selected lines.
- move the cursor to the new position and "`p`"ut the lines there.

It is sometimes difficult to see or remember where a fold is located, thus where a |`:h 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.

You can use the mouse to open a fold by clicking on the "+" in the foldcolumn. Clicking on the "-" or a "|" below it will close an open fold.

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

When in Insert mode, the fold at the cursor line is never closed. That allows you to see what you type!

Folds are opened automatically when jumping around or moving the cursor left or right. For example, the "0" command opens the fold under the cursor (if 'foldopen' contains "hor", which is the default). The 'foldopen' option can be changed to open folds for specific commands. If you want the line under the cursor always to be open, do this:

```
:set foldopen=all
```

Warning: You won't be able to move onto a closed fold then. You might want to use this only temporarily and then set it back to the default:

```
:set foldopen&
```

You can make folds close automatically when you move out of it:

```
:set foldclose=all
```

This will re-apply 'foldlevel' to all folds that don't contain the cursor. You have to try it out if you like how this feels. Use |:h zm| to fold more and |:h zr| to fold less (reduce folds).

The folding is local to the window. This allows you to open two windows on the same buffer, one with folds and one without folds. Or one with all folds closed and one with all folds open.

Saving and restoring folds

When you abandon a file (starting to edit another one), the state of the folds is lost. If you come back to the same file later, all manually opened and closed folds are back to their default. When folds have been created manually, all folds are gone! To save the folds use the |:h :mkview| command:

```
:mkview
```

This will store the settings and other things that influence the view on the file. You can change what is stored with the 'viewoptions' option. 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.

Folding by indent

Defining folds with |:h 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 |:h zm| and |:h zr| commands to fold more and reduce folding. It's easy to see on this example text:

```
This line is not indented
    This line is indented once
        This line is indented twice
        This line is indented twice
    This line is indented once
This line is not indented
    This line is indented once
    This line is indented once
```

Note that the relation between the amount of indent and the fold depth depends on the 'shiftwidth' option. Each 'shiftwidth' worth of indent adds one to the depth of the fold. This is called a fold level.

When you use the |:h zr| and |:h 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. |:h zM| does set 'foldlevel' to zero. The opposite command |:h 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:

1. 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.
2. By using |:h zo| and |:h 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.

This can be combined: You can first close most folds by using |:h zm| a few times and then open a specific fold with b!|:h zo|!. Or open all folds with |:h zR| and then close specific folds with |:h zc|.

But you cannot manually define folds when 'foldmethod' is "indent", as that would conflict with the relation between the indent and the fold level.

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

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;
    /* }}} */
}
/* }}} */
```

Notice that the folded line will display the text before the marker. This is very useful to tell what the fold contains.

It's quite annoying when the markers don't pair up correctly after moving some lines around. This can be avoided by using numbered markers. Example:

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

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

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

At every numbered marker a fold at the specified level begins. This will make any fold at a higher level stop here. You can just use numbered start markers to define all folds. Only when you want to explicitly stop a fold before another starts you need to add an end marker.

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

Folding by syntax

For each language Vim uses a different syntax file. This defines the colors for various items in the file. If you are reading this in Vim, in a terminal that supports colors, the colors you see are made with the "`help`" syntax file.

In the syntax files it is possible to add syntax items that have the "`fold`" argument. These define a fold region. This requires writing a syntax file and adding these items in it. That's not so easy to do. But once it's done, all folding happens automatically.

Here we'll assume you are using an existing syntax file. Then there is nothing more to explain. You can open and close folds as explained above. The folds will be created and deleted automatically when you edit the file.

More about folding by syntax in the reference manual: |`:h fold-syntax`|

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: |`:h fold-expr`|

Folding unchanged lines

This is useful when you set the `'diff'` option in the same window. The |`:h 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 |`:h fold-diff`|.

Which fold method to use?

All these possibilities make you wonder which method you should choose. Unfortunately, there is no golden rule. Here are some hints.

If there is a syntax file with folding for the language you are editing, that is probably the best choice. If there isn't one, you might try to write it. This requires a good knowledge of search patterns. It's not easy, but when it's working you will not have to define folds manually.

Typing commands to manually fold regions can be used for unstructured text. Then use the |`:h :mkview`| command to save and restore your folds.

The marker method requires you to change the file. If you are sharing the files with other people or you have to meet company standards, you might not be allowed to add them.

The main advantage of markers is that you can put them exactly where you want them. That avoids that a few lines are missed when you cut and paste folds. And you can add a comment about what is contained in the fold.

Folding by indent is something that works in many files, but not always very well. Use it when you can't use one of the other methods. However, it is very useful for outlining. Then you specifically use one `'shiftwidth'` for each nesting level.

Folding with expressions can make folds in almost any structured text. It is quite simple to specify, especially if the start and end of a fold can easily be recognized.

If you use the "`expr`" method to define folds, but they are not exactly how you want them, you could switch to the "`manual`" method. This will not remove the defined folds. Then you can delete or add folds manually.

29. 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.

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
```

"`ctags`" is a separate program. Most Unix systems already have it installed. If you do not have it yet, you can find Exuberant ctags here: http://ctags.sf.net.

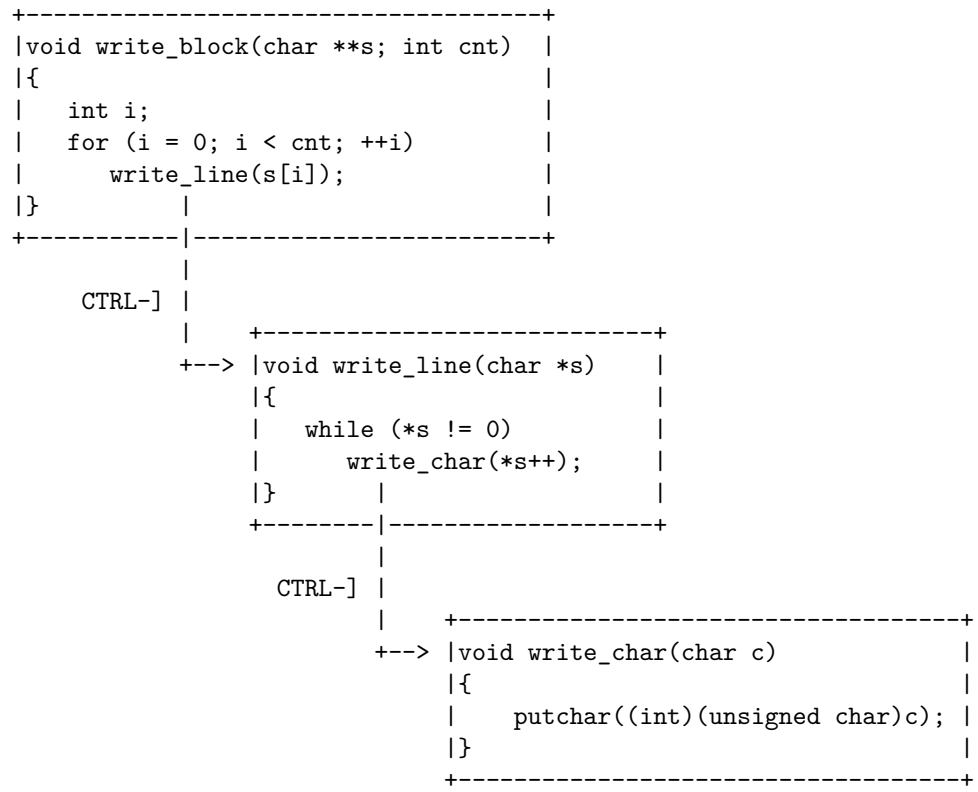
Now when you are in Vim and you want to go to a function definition, you can jump to it by using the following command:

```
: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. This makes it easy to explore a tangle of C code. Suppose, for example, that you are in the function "`write_block`". You can see that it calls "`write_line`". But what does "`write_line`" do? By placing the cursor on the call to "`write_line`" and pressing CTRL-], you jump to the definition of this function.

The "`write_line`" function calls "`write_char`". You need to figure out what it does. So you position the cursor over the call to "`write_char`" and press CTRL-]. Now you are at the definition of "`write_char`".



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

The ":tag" command replaces the file in the current window with the one containing the new function. But suppose you want to see not only the old function but also the new one? You can split the window using the ":split" command followed by the ":tag" command. Vim has a shorthand command that does both:

```
: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
```

Vim will present you with a list of choices:

```
  # pri kind tag               file
  1 F   f    mch_init          os_amiga.c
               mch_init()
  2 F   f    mch_init          os_mac.c
               mch_init()
  3 F   f    mch_init          os_msdos.c
               mch_init(void)
  4 F   f    mch_init          os_riscos.c
               mch_init()
Enter nr of choice (<CR> to abort):
```

You can now enter the number (in the first column) of the match that you would like to jump to. The information in the other columns give you a good idea of where the match is defined.

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

Command line completion is a good way to avoid typing a long tag name. Just type the first bit and press <Tab>:

```
:tag write_<Tab>
```

You will get the first match. If it's not the one you want, press <Tab> until you find the right one.

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_
```

The "`^`" specifies that the tag starts with "`write_`". Otherwise it would also be found halfway a tag name. Similarly "`$`" at the end makes sure the pattern matches until the end of a tag.

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 |`:h 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 |`:h cscope`|.

The preview window

When you edit code that contains a function call, you need to use the correct arguments. To know what values to pass you can look at how the function is defined. The tags mechanism works very well for this. Preferably the definition is displayed in another window. For this the preview window can be used.

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 |`:h 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.

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:

```
]#
```

These two commands skip any "`#if`" - "`#endif`" blocks that they encounter. Example:

```
#if defined(HAS_INC_H)
    a = a + inc();
# ifdef USE_THEME
    a += 3;
# endif
    set_width(a);
```

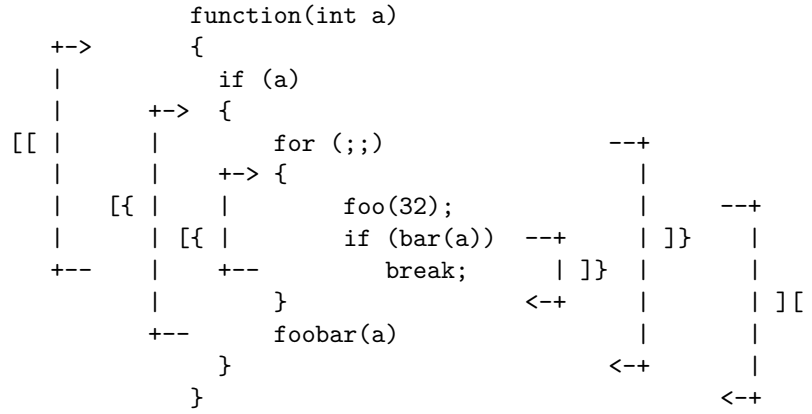
With the cursor in the last line, "`[#`" moves to the first line. The "`#ifdef`" - "`#endif`" block in the middle is skipped.

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.

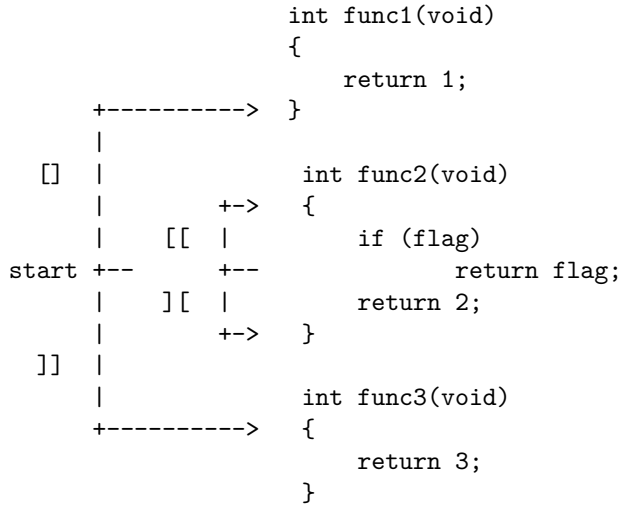
The "[{" command moves to the start of the current block. It skips over pairs of {} at the same level. "]}" jumps to the end.

An overview:



When writing C++ or Java, the outer {} block is for the class. The next level of {} is for a method. When somewhere inside a class use "[m" to find the previous start of a method. "]m" finds the next start of a method.

Additionally, "[]" moves backward to the end of a function and "]]" moves forward to the start of the next function. The end of a function is defined by a "}" in the first column.



Don't forget you can also use "%" to move between matching (), {} and []. That also works when they are many lines apart.

Moving in braces

The "[(" and "])" commands work similar to "[{" and "]}", except that they work on () pairs instead of {} pairs.

```
                  [(
      <--------------------------------
                 <-------
    if (a == b && (c == d || (e > f)) && x > y)
                       -------------->
              ---------------------------------->
                           ])
```

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.

```
   +->     +-> /*
   |    [/ |    * A comment about      --+
[/ |       +--  * wonderful life.        | ]/
   |            */                     <-+
   |
   +--          foo = bar * 3;         --+
                                         | ]/
                /* a short comment */  <-+
```

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.). The result looks like this:

```
structs.h
 1:   29     unsigned     column;    /* column number */
```

The advantage over using tags or the preview window is that included files are searched. In most cases this results in the right declaration to be found. Also when the tags file is out of date. Also when you don't have tags for the included files.

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. An example of the output:

```
--- Included files not found in path ---
<io.h>
vim.h -->
  <functions.h>
  <clib/exec_protos.h>
```

The "`io.h`" file is included by the current file and can't be found. "`vim.h`" can be found, thus "`:checkpath`" goes into this file and checks what it includes. The "`functions.h`" and "`clib/exec_protos.h`" files, included by "`vim.h`" are not found.

Note: Vim is not a compiler. It does not recognize "`#ifdef`" statements. This means every "`#include`" statement is used, also when it comes after "`#if NEVER`".

To fix the files that could not be found, add a directory to the `'path'` option. A good place to find out about this is the Makefile. Look out for lines that contain "`-I`" items, like "`-I/usr/local/X11`". To add this directory use:

```
:set path+=/usr/local/X11
```

When there are many subdirectories, you can use the "`*`" wildcard. Example:

```
:set path+=/usr/*/include
```

This would find files in "`/usr/local/include`" as well as "`/usr/X11/include`".

When working on a project with a whole nested tree of included files, the "`**`" items is useful. This will search down in all subdirectories. Example:

```
:set path+=/projects/invent/**/include
```

This will find files in the directories:

```
/projects/invent/include
/projects/invent/main/include
/projects/invent/main/os/include
etc.
```

There are even more possibilities. Check out the `'path'` option for info.

If you want to see which included files are actually found, use this command:

```
:checkpath!
```

You will get a (very long) list of included files, the files they include, and so on. To shorten the list a bit, Vim shows "`(Already listed)`" for files that were found before and doesn't list the included files in there again.

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

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). Example (cursor on "`counter`"):

```
      +->   static int counter = 0;
      |
      |     int get_counter(void)
   gD |     {
      |         ++counter;
      +--       return counter;
            }
```

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. Example (cursor on "`idx`"):

```
        int find_entry(char *name)
        {
   +->      int idx;
   |
gd |        for (idx = 0; idx < table_len; ++idx)
   |          if (strcmp(table[idx].name, name) == 0)
   +--             return idx;
        }
```

30. 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.

Compiling

Vim has a set of so called "quickfix" commands. They enable you to compile a program from within Vim and then go through the errors generated and fix them (hopefully). You can then recompile and fix any new errors that are found until finally your program compiles without any error.

The following command runs the program "make" (supplying it with any argument you give) and captures the results:

```
:make {arguments}
```

If errors were generated, they are captured and the editor positions you where the first error occurred.

Take a look at an example ":make" session. (Typical :make sessions generate far more errors and fewer stupid ones.) After typing ":make" the screen looks like this:

```
:!make | &tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function 'main':
main.c:6: too many arguments to function 'do_sub'
main.c: At top level:
main.c:10: parse error before '}'
make: *** [prog] Error 1

2 returned
"main.c" 11L, 111C
(3 of 6): too many arguments to function 'do_sub'
Press ENTER or type command to continue
```

From this you can see that you have errors in the file "main.c". When you press <Enter>, Vim displays the file "main.c", with the cursor positioned on line 6, the first line with an error. You did not need to specify the file or the line number, Vim knew where to go by looking in the error messages.

```
              +--------------------------------------------------+
              |int main()                                        |
              |{                                                 |
              |  int i=3;                                        |
   cursor -> |  do_sub("foo");                                  |
              |  ++i;                                            |
              |  return (0);                                     |
              |}                                                 |
              |}                                                 |
              | ~                                                |
              |(3 of 12): too many arguments to function 'do_sub' |
              +--------------------------------------------------+
```

The following command goes to where the next error occurs:

```
:cnext
```

Vim jumps to line 10, the last line in the file, where there is an extra '}'.

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. The output looks like this:

```
:clist
   3 main.c: 6:too many arguments to function 'do_sub'
   5 main.c: 10:parse error before '}'
```

Only the lines where Vim recognized a file name and line number are listed here. It assumes those are the interesting lines and the rest is just boring messages. However, sometimes unrecognized lines do contain something you want to see. Output from the linker, for example, about an undefined function. To see all the messages add a "!" to the command:

```
:clist!
   1 gcc -g -Wall -o prog main.c sub.c
   2 main.c: In function 'main':
   3 main.c:6: too many arguments to function 'do_sub'
   4 main.c: At top level:
   5 main.c:10: parse error before '}'
   6 make: *** [prog] Error 1
```

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

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: |`:h filename-modifiers`|.

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.

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: |`:h 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`|.

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.

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
```

With this option enabled, when you type something such as "`if (x)`", the next line will automatically be indented an additional level.

```
                          if (flag)
    Automatic indent   --->     do_the_work();
    Automatic unindent <--   if (other_flag) {
    Automatic indent   --->     do_file();
    keep indent                 do_some_more();
    Automatic unindent <--   }
```

When you type something in curly braces (`{}`), the text will be indented at the start and unindented at the end. The unindenting will happen after typing the '`}`', since Vim can't guess what you are going to type.

One side effect of automatic indentation is that it helps you catch errors in your code early. When you type a `}` to finish a function, only to find that the automatic indentation gives it more indent than what you expected, there is probably a `}` missing. Use the "`%`" command to find out which `{` matches the `}` you typed.

A missing `)` and `;` also cause extra indent. Thus if you get more white space than you would expect, check the preceding lines.

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 |`:h cinoptions-values`|.

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
```

Actually, this does a lot more than switching on `'cindent'` for C files. First of all, it enables detecting the type of a file. That's the same as what is used for syntax highlighting.

When the filetype is known, Vim will search for an indent file for this type of file. The Vim distribution includes a number of these for various programming languages. This indent file will then prepare for automatic indenting specifically for this file.

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: |`:h indent-expression`|.

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. This is useful for languages where no indent file is available. `'smartindent'` is not as smart as `'cindent'`, but smarter than `'autoindent'`.

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. In this example the cursor is on "`printf`":

original text	after "`>i{`"	after "`>a{`"

`if (flag)`
`{`
`printf("yes");`
`flag = 0;`
`}`
	`if (flag)`
	`{`
	`  printf("yes");`
	`  flag = 0;`
	`}`
		`if (flag)`
		`  {`
		`  printf("yes");`
		`  flag = 0;`
		`  }`

Tabs and spaces

`'tabstop'` is set to eight by default. Although you can change it, you quickly run into trouble later. Other programs won't know what tabstop value you used. They probably use the default value of eight, and your text suddenly looks very different. Also, most printers use a fixed tabstop value of eight. Thus it's best to keep `'tabstop'` alone. (If you edit a file which was written with a different tabstop setting, see |`Indents and tabs`| for how to fix that.)

For indenting lines in a program, using a multiple of eight spaces makes you quickly run into the right border of the window. Using a single space doesn't provide enough visual difference. Many people prefer to use four spaces, a good compromise.

Since a <Tab> is eight spaces and you want to use an indent of four spaces, you can't use a <Tab> character to make your indent. There are two ways to handle this:

1. Use a mix of <Tab> and space characters. Since a <Tab> takes the place of eight spaces, you have fewer characters in your file. Inserting a <Tab> is quicker than eight spaces. Backspacing works faster as well.
2. Use spaces only. This avoids the trouble with programs that use a different tabstop value.

Fortunately, Vim supports both methods quite well.

Spaces and tabs

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.

The following shows what happens pressing <Tab> a few times, and then using <BS>. A "`.`" stands for a space and "`------->`" for a <Tab>.

type	result
`<Tab>`	`....`
`<Tab><Tab>`	`------->`
`<Tab><Tab><Tab>`	`------->....`
`<Tab><Tab><Tab><BS>`	`------->`
`<Tab><Tab><Tab><BS><BS>`	`....`

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. Thus you get the same amount of white space as if a <Tab> character was inserted, but there isn't a real <Tab> character in your file.

The backspace key will delete each space by itself. Thus after typing one <Tab> you have to press the <BS> key up to eight times to undo it. If you are in the indent, pressing CTRL-D will be a lot quicker.

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!
```

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.
 */
```

Notice that Vim properly handled the beginning of each line. An alternative is to select the text that is to be formatted in Visual mode and type "`gq`".

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 |`:h fo-table`| for more flags.

Defining a comment

The `'comments'` option defines what a comment looks like. Vim distinguishes between a single-line comment and a comment that has a different start, end and middle part.

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 |:h format-comments|.

31. 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.

The file browser

When using the File/Open... menu you get a file browser. This makes it easier to find the file you want to edit. But what if you want to split a window to edit another file? There is no menu entry for this. You could first use Window/Split and then File/Open..., but that's more work.

Since you are typing most commands in Vim, opening the file browser with a typed command is possible as well. To make the split command use the file browser, prepend "`browse`":

```
:browse split
```

Select a file and then the "`:split`" command will be executed with it. If you cancel the file dialog nothing happens, the window isn't split.

You can also specify a file name argument. This is used to tell the file browser where to start. Example:

```
:browse split /etc
```

The file browser will pop up, starting in the directory "`/etc`".

The "`:browse`" command can be prepended to just about any command that opens a file.

If no directory is specified, Vim will decide where to start the file browser. By default it uses the same directory as the last time. Thus when you used "`:browse split`" and selected a file in "`/usr/local/share`", the next time you use a "`:browse`" it will start in "`/usr/local/share`" again.

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`".

Note: To avoid using the mouse, most file browsers offer using key presses to navigate. Since this is different for every system, it is not explained here. Vim uses a standard browser when possible, your system documentation should contain an explanation on the keyboard shortcuts somewhere.

When you are not using the GUI version, you could use the file explorer window to select files like in a file browser. However, this doesn't work for the "`:browse`" command. See |`:h netrw-browse`|.

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
```

Vim will pop up a dialog that looks something like this:

```
+-------------------------------------+
|                                     |
|   ?  Save changes to "bar.txt"?     |
|                                     |
|   YES   NO        CANCEL            |
+-------------------------------------+
```

Now make your choice. If you do want to save the changes, select "YES". If you want to lose the changes for ever: "NO". If you forgot what you were doing and want to check what really changed use "CANCEL". You will be back in the same file, with the changes still there.

Just like "`:browse`", the "`:confirm`" command can be prepended to most commands that edit another file. They can also be combined:

```
:confirm browse edit
```

This will produce a dialog when the current buffer was changed. Then it will pop up a file browser to select the file to edit.

Note: In the dialog you can use the keyboard to select the choice. Typically the <Tab> key and the cursor keys change the choice. Pressing <Enter> selects the choice. This depends on the system though.

When you are not using the GUI, the "`:confirm`" command works as well. Instead of popping up a dialog, Vim will print the message at the bottom of the Vim window and ask you to press a key to make a choice.

```
:confirm edit main.c
   Save changes to "Untitled"?
   [Y]es, (N)o, (C)ancel:
```

You can now press the single key for the choice. You don't have to press <Enter>, unlike other typing on the command line.

Menu shortcuts

The keyboard is used for all Vim commands. The menus provide a simple way to select commands, without knowing what they are called. But you have to move your hand from the keyboard and grab the mouse.

Menus can often be selected with keys as well. This depends on your system, but most often it works this way. Use the <Alt> key in combination with the underlined letter of a menu. For example, <A-w> (<Alt> and w) pops up the Window menu.

In the Window menu, the "`split`" item has the p underlined. To select it, let go of the <Alt> key and press p.

After the first selection of a menu with the <Alt> key, you can use the cursor keys to move through the menus. <Right> selects a submenu and <left> closes it. <Esc> also closes a menu. <Enter> selects a menu item.

There is a conflict between using the <Alt> key to select menu items, and using <Alt> key combinations for mappings. The `'winaltkeys'` option tells Vim what it should do with the <Alt> key.

The default value "`menu`" is the smart choice: If the key combination is a menu shortcut it can't be mapped. All other keys are available for mapping.

The value "`no`" doesn't use any <Alt> keys for the menus. Thus you must use the mouse for the menus, and all <Alt> keys can be mapped.

The value "`yes`" means that Vim will use any <Alt> keys for the menus. Some <Alt> key combinations may also do other things than selecting a menu.

Vim window position and size

To see the current Vim window position on the screen use:

```
:winpos
```

This will only work in the GUI. The output may look like this:

```
Window position: X 272, Y 103
```

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
```

Note: There may be a small offset between the reported position and where the window moves. This is because of the border around the window. This is added by the window manager.

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. Thus this depends on the size of the font being used. You can see the current size with this command:

```
: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
```

Various

You can use gvim to edit an e-mail message. In your e-mail program you must select gvim to be the editor for messages. When you try that, you will see that it doesn't work: The mail program thinks that editing is finished, while gvim is still running!

What happens is that gvim disconnects from the shell it was started in. That is fine when you start gvim in a terminal, so that you can do other work in that terminal. But when you really want to wait for gvim to finish, you must prevent it from disconnecting. The "`-f`" argument does this:

```
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.

Delayed start of the gui

On Unix it's possible to first start Vim in a terminal. That's useful if you do various tasks in the same shell. If you are editing a file and decide you want to use the GUI after all, you can start it with:

```
:gui
```

Vim will open the GUI window and no longer use the terminal. You can continue using the terminal for something else. The "`-f`" argument is used here to run the GUI in the foreground. You can also use "`:gui -f`".

The gvim startup file

When gvim starts, it reads the gvimrc file. That's similar to the vimrc file used when starting Vim. The gvimrc file can be used for settings and commands that are only to be used when the GUI is going to be started. For example, you can set the `'lines'` option to set a different window size:

```
:set lines=55
```

You don't want to do this in a terminal, since its size is fixed (except for an xterm that supports resizing).

The gvimrc file is searched for in the same locations as the vimrc file. Normally its name is "`~/.gvimrc`" for Unix and "`$VIM/_gvimrc`" for MS-Windows. The `$MYGVIMRC` environment variable is set to it, thus you can use this command to edit the file, if you have one:

```
:edit $MYGVIMRC
```

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 ...
```

That allows starting gvim for different kinds of editing. You could set another font size, for example.

To completely skip reading a gvimrc file:

```
gvim -U NONE ...
```

32. 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.

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 |`:h undo-persistence`|.

Numbering changes

In section |`Undo and Redo`| 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 |:h u| command to undo. If you do this twice you get to "one". Use |:h CTRL-R| to redo, and you will go to "one too". One more |:h 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.

What matters here is the order in which the changes are made. Undo and redo are not considered changes in this context. After each change you have a new state of the text.

Note that only the changes are numbered, the text shown in the tree above has no identifier. They are mostly referred to by the number of the change above it. But sometimes by the number of one of the changes below it, especially when moving up in the tree, so that you know which change was just undone.

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 |:h :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 |:h 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 |:h g-| and you will see the text change:

```
me too
one two
one too
one
```

Use |:h g+| to move forward in time:

```
one
one too
one two
me too
not two
```

Using |:h :undo| is useful if you know what change you want to jump to. |g-| and |:h g+| are useful if you don't know exactly what the change number is.

You can type a count before |:h g-| and |:h g+| to repeat them.

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 |:h :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 |:h \verb:later!!| command:

```
:later 1m
```

The arguments are "s", "m" and "h", just like with |:h :earlier|.

If you want even more details, or want to manipulate the information, you can use the |:h undotree()| function. To see what it returns:

```
:echo undotree()
```

40. 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.

Key mapping

A simple mapping was explained in section |`Simple mappings`|. The principle is that one sequence of key strokes is translated into another sequence of key strokes. This is a simple, yet powerful mechanism.

The simplest form is that one key is mapped to a sequence of keys. Since the function keys, except <F1>, have no predefined meaning in Vim, these are good choices to map. Example:

```
:map <F2> GoDate: <Esc>:read !date<CR>kJ
```

This shows how three modes are used. After going to the last line with "`G`", the "`o`" command opens a new line and starts Insert mode. The text "`Date: `" is inserted and <Esc> takes you out of insert mode.

Notice the use of special keys inside <>. This is called angle bracket notation. You type these as separate characters, not by pressing the key itself. This makes the mappings better readable and you can copy and paste the text without problems.

The "`:`" character takes Vim to the command line. 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.

At this point of execution the text looks like this:

```
Date:
Fri Jun 15 12:54:34 CEST 2001
```

Now "`kJ`" moves the cursor up and joins the lines together.

To decide which key or keys you use for mapping, see |`:h map-which-keys`|.

Mapping and modes

The "`:map`" command defines remapping for keys in Normal mode. You can also define mappings for other modes. For example, "`:imap`" applies to Insert mode. You can use it to insert a date below the cursor:

```
:imap <F2> <CR>Date: <Esc>:read !date<CR>kJ
```

It looks a lot like the mapping for <F2> in Normal mode, only the start is different. The <F2> mapping for Normal mode is still there. Thus you can map the same key differently for each mode.

Notice that, although this mapping starts in Insert mode, it ends in Normal mode. If you want it to continue in Insert mode, append an "`a`" to the mapping.

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{
```

This causes <F7> to perform a select block "a{" in operator-pending mode, just like you typed it. This mapping is useful if typing a { on your keyboard is a bit difficult.

Listing mappings

To see the currently defined mappings, use ":map" without arguments. Or one of the variants that include the mode in which they work. The output could look like this:

```
    _g        :call MyGrep(1)<CR>
v   <F2>      :s/^/> /<CR>:noh<CR>``
n   <F2>      :.,$s/^/> /<CR>:noh<CR>``
    <xHome>   <Home>
    <xEnd>    <End>
```

The first column of the list shows in which mode the mapping is effective. This is "n" for Normal mode, "i" for Insert mode, etc. A blank is used for a mapping defined with ":map", thus effective in both Normal and Visual mode.

One useful purpose of listing the mapping is to check if special keys in <> form have been recognized (this only works when color is supported). For example, when <Esc> is displayed in color, it stands for the escape character. When it has the same color as the other text, it is five characters.

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>,,
```

Now you type ",,". This triggers the mapping. It replaces "5.1" with "5.2" in the first line. Then it does a ":wnext" to write the file and edit the next one. The mapping ends in ",,". This triggers the same mapping again, thus doing the substitution, etc.

This continues until there is an error. In this case it could be a file where the substitute command doesn't find a match for "5.1". You can then make a change to insert "5.1" and continue by typing ",," again. Or the ":wnext" fails, because you are in the last file in the list.

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 |:h :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

Abbreviations are a lot like Insert mode mappings. The arguments are handled in the same way. The main difference is the way they are triggered. An abbreviation is triggered by typing a non-word character after the word. A mapping is triggered when typing the last character.

Another difference is that the characters you type for an abbreviation are inserted in the text while you type them. When the abbreviation is triggered these characters are deleted and replaced by what the abbreviation produces. When typing the characters for a mapping, nothing is inserted until you type the last character that triggers it. If the 'showcmd' option is set, the typed characters are displayed in the last line of the Vim window.

An exception is when a mapping is ambiguous. Suppose you have done two mappings:

```
:imap aa foo
:imap aaa bar
```

Now, when you type "`aa`", Vim doesn't know if it should apply the first or the second mapping. It waits for another character to be typed. If it is an "`a`", the second mapping is applied and results in "`bar`". If it is a space, for example, the first mapping is applied, resulting in "`foo`", and then the space is inserted.

Additionally...

The <script> keyword can be used to make a mapping local to a script. See |`:h :map-<script>`|.

The <buffer> keyword can be used to make a mapping local to a specific buffer. See |`:h :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 |`:h :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>.

Defining command-line commands

The Vim editor enables you to define your own 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.

Note: User-defined commands must start with a capital letter. You cannot use "`:X`", "`:Next`" and "`:Print`". The underscore cannot be used! You can use digits, but this is discouraged.

To list the user-defined commands, execute the following command:

```
:command
```

Just like with the builtin commands, the user defined commands can be abbreviated. You need to type just enough to distinguish the command from another. Command line completion can be used to get the full name.

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 t
`-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 availa
`-complete={type}`	Type of command-line completion used. See |`:h :command-completion`| for the list of possible value
`-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: \|`:h user-commands`\|.

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 \|`:h 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 |:h 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 GNew 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).

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 |`:h 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=
```

41. 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.

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.

Let's start with a simple example:

```
:let i = 1
:while i < 5
:  echo "count is" i
:  let i += 1
:endwhile
```

Note: The ":" characters are not really needed here. You only need to use them when you type a command. In a Vim script file they can be left out. We will use them here anyway to make clear these are colon commands and make them stand out from Normal mode commands. *Note*: You can try out the examples by yanking the lines from the text here and executing them with `:@"`

The output of the example code is:

```
count is 1
count is 2
count is 3
count is 4
```

In the first line the "`:let`" command assigns a value to a variable. The generic form is:

```
:let {variable} = {expression}
```

In this case the variable name is "`i`" and the expression is a simple value, the number one.

The "`:while`" command starts a loop. The generic form is:

```
:while {condition}
:  {statements}
:endwhile
```

The statements until the matching "`:endwhile`" are executed for as long as the condition is true. The condition used here is the expression "`i < 5`". This is true when the variable i is smaller than five.

Note: If you happen to write a while loop that keeps on running, you can interrupt it by pressing CTRL-C (CTRL-Break on MS-Windows).

The "`:echo`" command prints its arguments. In this case the string "count is" and the value of the variable i. Since i is one, this will print:

```
count is 1
```

Then there is the "`:let i += 1`" command. This does the same thing as "`:let i = i + 1`". This adds one to the variable i and assigns the new value to the same variable.

The example was given to explain the commands, but would you really want to make such a loop it can be written much more compact:

```
:for i in range(1, 4)
:  echo "count is" i
:endfor
```

We won't explain how |`:h :for`| and |`:h range()`| work until later. Follow the links if you are impatient.

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. Careful: don't put a zero before a decimal number, it will be interpreted as an octal number!

The "`:echo`" command always prints decimal numbers. Example:

```
:echo 0x7f 036
127 30
```

A number is made negative with a minus sign. This also works for hexadecimal and octal numbers. A minus sign is also used for subtraction. Compare this with the previous example:

```
:echo 0x7f -036
97
```

White space in an expression is ignored. However, it's recommended to use it for separating items, to make the expression easier to read. For example, to avoid the confusion with a negative number above, put a space between the minus sign and the following number:

```
:echo 0x7f - 036
```

Variables

A variable name consists of ASCII letters, digits and the underscore. It cannot start with a digit. Valid variable names are:

```
counter
_aap3
very_long_variable_name_with_underscores
FuncLength
LENGTH
```

Invalid names are "`foo+bar`" and "`6var`".

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
```

Since "`s:count`" is local to this script, you can be sure that sourcing the "`other.vim`" script will not change this variable. If "`other.vim`" also uses an "`s:count`" variable, it will be a different copy, local to that script. More about script-local variables here: |`:h script-variable`|.

There are more kinds of variables, see |`:h 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

Variables take up memory and show up in the output of the "`:let`" command. To delete a variable use the "`:unlet`" command. Example:

```
: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. Example:

```
:if !exists("\verb!s:call_count!")
:  let s:call_count = 0
:endif
:let s:call_count = s:call_count + 1
:echo "called" s:call_count "times"
```

The "`exists()`" function checks if a variable has already been defined. Its argument is the name of the variable you want to check. Not the variable itself! If you would do this:

```
:if !exists(s:call_count)
```

Then the value of `s:call_count` will be used as the name of the variable that `exists()` checks. That's not what you want.

The exclamation mark `!` negates a value. When the value was true, it becomes false. When it was false, it becomes true. You can read it as "not". Thus "`if !exists()`" can be read as "`if not exists()`".

What Vim calls true is anything that is not zero. Zero is false.

Note: Vim automatically converts a string to a number when it is looking for a number. When using a string that doesn't start with a digit the resulting number is zero. Thus look out for this:

```
  :if "true"
```

The "`true`" will be interpreted as a zero, thus as false!

String variables and constants

So far only numbers were used for the variable value. Strings can be used as well. 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 |`Lists and Dictionaries`|.

To assign a string value to a variable, you need to use a string constant. There are two types of these. First the string in double quotes:

```
:let name = "peter"
:echo name
   peter
```

If you want to include a double quote inside the string, put a backslash in front of it:

```
:let name = "\"peter\""
:echo name
   "peter"
```

To avoid the need for a backslash, you can use a string in single quotes:

```
:let name = '"peter"'
:echo name
   "peter"
```

Inside a single-quote string all the characters are as they are. Only the single quote itself is special: you need to use two to get one. A backslash is taken literally, thus you can't use it to change the meaning of the character after it.

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 |`:h expr-quote`| for the full list of special items in a string.

Expressions

Vim has a rich, yet simple way to handle expressions. You can read the definition here: |`:h expression-syntax`|. Here we will show the most common items.

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

Examples:

```
:echo "The value of 'tabstop' is" &ts
:echo "Your home directory is" $HOME
:if @a > 5
```

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 |`:h /\C`|.)

Mathematics

It becomes more interesting if we combine these basic items. Let's start with mathematics on numbers:

`a + b`	add
`a - b`	subtract
`a * b`	multiply
`a / b`	divide
`a % b`	modulo

The usual precedence is used. Example:

```
:echo 10 + 5 * 2
   20
```

Grouping is done with parentheses. No surprises here. Example:

```
:echo (10 + 5) * 2
   30
```

Strings can be concatenated with "`.`". Example:

```
:echo "foo" . "bar"
   foobar
```

When the "`:echo`" command gets multiple arguments, it separates them with a space. In the example the argument is a single expression, thus no space is inserted.

Borrowed from the C language is the conditional expression:

```
a ? b : c
```

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
```

The three parts of the constructs are always evaluated first, thus you could see it work as:

```
(a) ? (b) : (c)
```

Conditionals

The "`:if`" commands executes the following statements, until the matching "`:endif`", only when a condition is met. The generic form is:

```
:if {condition}
   {statements}
:endif
```

Only when the expression `{condition}` evaluates to true (non-zero) will the `{statements}` be executed. These must still be valid commands. If they contain garbage, Vim won't be able to find the "`:endif`".

You can also use "`:else`". The generic form for this is:

```
:if {condition}
   {statements}
:else
   {statements}
:endif
```

The second `{statements}` is only executed if the first one isn't.

Finally, there is "`:elseif`":

```
:if {condition}
   {statements}
:elseif {condition}
   {statements}
:endif
```

This works just like using "`:else`" and then "`if`", but without the need for an extra "`:endif`".

A useful example for your vimrc file is checking the `'term'` option and doing something depending upon its value:

```
:if &term == "xterm"
:  " Do stuff for xterm
:elseif &term == "vt100"
:  " Do stuff for a vt100 terminal
:else
:  " Do something for other terminals
:endif
```

Logic operations

We already used some of them in the examples. These are the most often used ones:

`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 result is one if the condition is met and zero otherwise. An example:

```
:if v:version >= 700
:  echo "congratulations"
:else
:  echo "you are using an old version, upgrade!"
:endif
```

Here "`v:version`" is a variable defined by Vim, which has the value of the Vim version. 600 is for version 6.0. Version 6.1 has the value 601. This is very useful to write a script that works with multiple versions of Vim. |`:h v:version`|

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.

When comparing a string with a number, the string is first converted to a number. This is a bit tricky, because when a string doesn't look like a number, the number zero is used. Example:

```
:if 0 == "one"
:  echo "yes"
:endif
```

This will echo "`yes`", because "`one`" doesn't look like a number, thus it is converted to the number zero.

For strings there are two more items:

`a =~ b`	matches with
`a !~ b`	does not match with

The left item "`a`" is used as a string. The right item "`b`" is used as a pattern, like what's used for searching. Example:

```
:if str =~ " "
:  echo "str contains a space"
:endif
:if str !~ '\.$'
:  echo "str does not end in a full stop"
:endif
```

Notice the use of a single-quote string for the pattern. This is useful, because backslashes would need to be doubled in a double-quote string and patterns tend to contain many backslashes.

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 |`:h 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.

Example:

```
:while counter < 40
:  call do_something()
:  if skip_flag
:    continue
:  endif
:  if finished_flag
:    break
:  endif
:  sleep 50m
:endwhile
```

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 |Lists and Dictionaries|.

Executing an expression

So far the commands in the script were executed by Vim directly. The ":execute" command allows executing the result of an expression. This is a very powerful way to build commands and execute them.

An example is to jump to a tag, which is contained in a variable:

```
: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
```

The variable "normal_commands" must contain the Normal mode commands.

Make sure that the argument for ":normal" is a complete command. Otherwise Vim will run into the end of the argument and abort the command. For example, if you start Insert mode, you must leave Insert mode as well. This works:

```
:execute "normal Inew text \<Esc>"
```

This inserts "new text " in the current line. 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
```

Using functions

Vim defines many functions and provides a large amount of functionality that way. A few examples will be given in this section. You can find the whole list here: |:h functions|.

A function is called with the ":call" command. The parameters are passed in between parentheses separated by commas. Example:

```
:call search("Date: ", "W")
```

This calls the search() function, with arguments "Date: " and "W". The search() function uses its first argument as a search pattern and the second one as flags. The "W" flag means the search doesn't wrap around the end of the file.

A function can be called in an expression. Example:

```
:let line = getline(".")
:let repl = substitute(line, '\a', "*", "g")
:call setline(".", repl)
```

The getline() function obtains a line from the current buffer. Its argument is a specification of the line number. In this case "." is used, which means the line where the cursor is.

The substitute() function does something similar to the ":substitute" command. The first argument is the string on which to perform the substitution. The second argument is the pattern, the third the replacement string. Finally, the last arguments are the flags.

The setline() function sets the line, specified by the first argument, to a new string, the second argument. In this example the line under the cursor is replaced with the result of the substitute(). Thus the effect of the three statements is equal to:

```
:substitute/\a/*/g
```

Using the functions becomes more interesting when you do more work before and after the substitute() call.

Functions

There are many functions. We will mention them here, grouped by what they are used for. You can find an alphabetical list here: |:h functions|. Use CTRL-] on the function name to jump to detailed help on it.

String manipulation:

`nr2char()`	get a character by its ASCII value
`char2nr()`	get ASCII value of a character
`str2nr()`	convert a string to a Number
`str2float()`	convert a string to a Float
`printf()`	format a string according to `%` items
`escape()`	escape characters in a string with a '\'
`shellescape()`	escape a string for use with a shell command
`fnameescape()`	escape a file name for use with a Vim command
`tr()`	translate characters from one set to another
`strtrans()`	translate a string to make it printable
`tolower()`	turn a string to lowercase
`toupper()`	turn a string to uppercase
`match()`	position where a pattern matches in a string
`matchend()`	position where a pattern match ends in a string
`matchstr()`	match of a pattern in a string
`matchlist()`	like `matchstr()` and also return submatches
`stridx()`	first index of a short string in a long string
`strridx()`	last index of a short string in a long string
`strlen()`	length of a string
`substitute()`	substitute a pattern match with a string
`submatch()`	get a specific match in a "`:substitute`"
`strpart()`	get part of a string
`expand()`	expand special keywords
`iconv()`	convert text from one encoding to another
`byteidx()`	byte index of a character in a string
`repeat()`	repeat a string multiple times
`eval()`	evaluate a string expression

List manipulation:

`get()`	get an item without error for wrong index
`len()`	number of items in a List
`empty()`	check if List is empty
`insert()`	insert an item somewhere in a List
`add()`	append an item to a List
`extend()`	append a List to a List
`remove()`	remove one or more items from a List
`copy()`	make a shallow copy of a List
`deepcopy()`	make a full copy of a List
`filter()`	remove selected items from a List
`map()`	change each List item
`sort()`	sort a List
`reverse()`	reverse the order of a List
`split()`	split a String into a List
`join()`	join List items into a String
`range()`	return a List with a sequence of numbers
`string()`	String representation of a List
`call()`	call a function with List as arguments
`index()`	index of a value in a List
`max()`	maximum value in a List
`min()`	minimum value in a List
`count()`	count number of times a value appears in a List
`repeat()`	repeat a List multiple times

Dictionary manipulation:

`get()`	get an entry without an error for a wrong key
`len()`	number of entries in a Dictionary
`has_key()`	check whether a key appears in a Dictionary
`empty()`	check if Dictionary is empty
`remove()`	remove an entry from a Dictionary
`extend()`	add entries from one Dictionary to another
`filter()`	remove selected entries from a Dictionary
`map()`	change each Dictionary entry
`keys()`	get List of Dictionary keys
`values()`	get List of Dictionary values
`items()`	get List of Dictionary key-value pairs
`copy()`	make a shallow copy of a Dictionary
`deepcopy()`	make a full copy of a Dictionary
`string()`	String representation of a Dictionary
`max()`	maximum value in a Dictionary
`min()`	minimum value in a Dictionary
`count()`	count number of times a value appears

Floating point computation:

`float2nr()`	convert Float to Number
`abs()`	absolute value (also works for Number)
`round()`	round off
`ceil()`	round up
`floor()`	round down
`trunc()`	remove value after decimal point
`log10()`	logarithm to base 10
`pow()`	value of x to the exponent y
`sqrt()`	square root
`sin()`	sine
`cos()`	cosine
`tan()`	tangent
`asin()`	arc sine
`acos()`	arc cosine
`atan()`	arc tangent
`atan2()`	arc tangent
`sinh()`	hyperbolic sine
`cosh()`	hyperbolic cosine
`tanh()`	hyperbolic tangent

Variables:

`type()`	type of a variable
`islocked()`	check if a variable is locked
`function()`	get a Funcref for a function name
`getbufvar()`	get a variable value from a specific buffer
`setbufvar()`	set a variable in a specific buffer
`getwinvar()`	get a variable from specific window
`gettabvar()`	get a variable from specific tab page
`gettabwinvar()`	get a variable from specific window & tab page
`setwinvar()`	set a variable in a specific window
`settabvar()`	set a variable in a specific tab page
`settabwinvar()`	set a variable in a specific window & tab page
`garbagecollect()`	possibly free memory

Cursor and mark position:

`col()`	column number of the cursor or a mark
`virtcol()`	screen column of the cursor or a mark
`line()`	line number of the cursor or mark
`wincol()`	window column number of the cursor
`winline()`	window line number of the cursor
`cursor()`	position the cursor at a line/column
`getpos()`	get position of cursor, mark, etc.
`setpos()`	set position of cursor, mark, etc.
`byte2line()`	get line number at a specific byte count
`line2byte()`	byte count at a specific line
`diff_filler()`	get the number of filler lines above a line

Working with text in the current buffer:

`getline()`	get a line or list of lines from the buffer
`setline()`	replace a line in the buffer
`append()`	append line or list of lines in the buffer
`indent()`	indent of a specific line
`cindent()`	indent according to C indenting
`lispindent()`	indent according to Lisp indenting
`nextnonblank()`	find next non-blank line
`prevnonblank()`	find previous non-blank line
`search()`	find a match for a pattern
`searchpos()`	find a match for a pattern
`searchpair()`	find the other end of a start/skip/end
`searchpairpos()`	find the other end of a start/skip/end
`searchdecl()`	search for the declaration of a name

System functions and manipulation of files:

`glob()`	expand wildcards
`globpath()`	expand wildcards in a number of directories
`findfile()`	find a file in a list of directories
`finddir()`	find a directory in a list of directories
`resolve()`	find out where a shortcut points to
`fnamemodify()`	modify a file name
`pathshorten()`	shorten directory names in a path
`simplify()`	simplify a path without changing its meaning
`executable()`	check if an executable program exists
`filereadable()`	check if a file can be read
`filewritable()`	check if a file can be written to
`getfperm()`	get the permissions of a file
`getftype()`	get the kind of a file
`isdirectory()`	check if a directory exists
`getfsize()`	get the size of a file
`getcwd()`	get the current working directory
`haslocaldir()`	check if current window used |`:h :lcd`|
`tempname()`	get the name of a temporary file
`mkdir()`	create a new directory
`delete()`	delete a file
`rename()`	rename a file
`system()`	get the result of a shell command
`hostname()`	name of the system
`readfile()`	read a file into a List of lines
`writefile()`	write a List of lines into a file

Date and Time:

`getftime()`	get last modification time of a file
`localtime()`	get current time in seconds
`strftime()`	convert time to a string
`reltime()`	get the current or elapsed time accurately
`reltimestr()`	convert reltime() result to a string

Buffers, windows and the argument list:

`argc()`	number of entries in the argument list
`argidx()`	current position in the argument list
`argv()`	get one entry from the argument list
`bufexists()`	check if a buffer exists
`buflisted()`	check if a buffer exists and is listed
`bufloaded()`	check if a buffer exists and is loaded
`bufname()`	get the name of a specific buffer
`bufnr()`	get the buffer number of a specific buffer
`tabpagebuflist()`	return List of buffers in a tab page
`tabpagenr()`	get the number of a tab page
`tabpagewinnr()`	like winnr() for a specified tab page
`winnr()`	get the window number for the current window
`bufwinnr()`	get the window number of a specific buffer
`winbufnr()`	get the buffer number of a specific window
`getbufline()`	get a list of lines from the specified buffer

Command line:

`getcmdline()`	get the current command line
`getcmdpos()`	get position of the cursor in the command line
`setcmdpos()`	set position of the cursor in the command line
`getcmdtype()`	return the current command-line type

Quickfix and location lists:

`getqflist()`	list of quickfix errors
`setqflist()`	modify a quickfix list
`getloclist()`	list of location list items
`setloclist()`	modify a location list

Insert mode completion:

`complete()`	set found matches
`complete_add()`	add to found matches
`complete_check()`	check if completion should be aborted
`pumvisible()`	check if the popup menu is displayed

Folding:

`foldclosed()`	check for a closed fold at a specific line
`foldclosedend()`	like foldclosed() but return the last line
`foldlevel()`	check for the fold level at a specific line
`foldtext()`	generate the line displayed for a closed fold
`foldtextresult()`	get the text displayed for a closed fold

Syntax and highlighting:

`clearmatches()`	clear all matches defined by |`:h matchadd()`| and the |`:h :match`| commands
`getmatches()`	get all matches defined by |`:h matchadd()`| and the |`:h :match`| commands
`hlexists()`	check if a highlight group exists
`hlID()`	get ID of a highlight group
`synID()`	get syntax ID at a specific position
`synIDattr()`	get a specific attribute of a syntax ID
`synIDtrans()`	get translated syntax ID
`synstack()`	get list of syntax IDs at a specific position
`synconcealed()`	get info about concealing
`diff_hlID()`	get highlight ID for diff mode at a position
`matchadd()`	define a pattern to highlight (a "match")
`matcharg()`	get info about |`:h :match`| arguments
`matchdelete()`	delete a match defined by |`:h matchadd()`| or a |`:h :match`| command
`setmatches()`	restore a list of matches saved by |`:h getmatches()`|

Spelling:

`spellbadword()`	locate badly spelled word at or after cursor
`spellsuggest()`	return suggested spelling corrections
`soundfold()`	return the sound-a-like equivalent of a word

History:

`histadd()`	add an item to a history
`histdel()`	delete an item from a history
`histget()`	get an item from a history
`histnr()`	get highest index of a history list

Interactive:

`browse()`	put up a file requester
`browsedir()`	put up a directory requester
`confirm()`	let the user make a choice
`getchar()`	get a character from the user
`getcharmod()`	get modifiers for the last typed character
`feedkeys()`	put characters in the typeahead queue
`input()`	get a line from the user
`inputlist()`	let the user pick an entry from a list
`inputsecret()`	get a line from the user without showing it
`inputdialog()`	get a line from the user in a dialog
`inputsave()`	save and clear typeahead
`inputrestore()`	restore typeahead

GUI:

`getfontname()`	get name of current font being used
`getwinposx()`	X position of the GUI Vim window
`getwinposy()`	Y position of the GUI Vim window

Vim server:

`serverlist()`	return the list of server names
`remote_send()`	send command characters to a Vim server
`remote_expr()`	evaluate an expression in a Vim server
`server2client()`	send a reply to a client of a Vim server
`remote_peek()`	check if there is a reply from a Vim server
`remote_read()`	read a reply from a Vim server
`foreground()`	move the Vim window to the foreground
`remote_foreground()`	move the Vim server window to the foreground

Window size and position:

`winheight()`	get height of a specific window
`winwidth()`	get width of a specific window
`winrestcmd()`	return command to restore window sizes
`winsaveview()`	get view of current window
`winrestview()`	restore saved view of current window

Various:

`mode()`	get current editing mode
`visualmode()`	last visual mode used
`hasmapto()`	check if a mapping exists
`mapcheck()`	check if a matching mapping exists
`maparg()`	get rhs of a mapping
`exists()`	check if a variable, function, etc. exists
`has()`	check if a feature is supported in Vim
`changenr()`	return number of most recent change
`cscope_connection()`	check if a cscope connection exists
`did_filetype()`	check if a FileType autocommand was used
`eventhandler()`	check if invoked by an event handler
`getpid()`	get process ID of Vim
`libcall()`	call a function in an external library
`libcallnr()`	idem, returning a number
`getreg()`	get contents of a register
`getregtype()`	get type of a register
`setreg()`	set contents and type of a register
`taglist()`	get list of matching tags
`tagfiles()`	get a list of tags files
`mzeval()`	evaluate |`:h MzScheme`| expression

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.

Let's define a short function to return the smaller of two numbers. It starts with this line:

```
:function Min(num1, num2)
```

This tells Vim that the function is named "`Min`" and it takes two arguments: "`num1`" and "`num2`".

The first thing you need to do is to check to see which number is smaller:

```
:  if a:num1 < a:num2
```

The special prefix "`a:`" tells Vim that the variable is a function argument. Let's assign the variable "`smaller`" the value of the smallest number:

```
:  if a:num1 < a:num2
:    let smaller = a:num1
:  else
:    let smaller = a:num2
:  endif
```

The variable "`smaller`" is a local variable. Variables used inside a function are local unless prefixed by something like "`g:`", "`a:`", or "`s:`".

Note: To access a global variable from inside a function you must prepend "`g:`" to it. Thus "`g:today`" inside a function is used for the global variable "`today`", and "`today`" is another variable, local to the function.

You now use the ":return" statement to return the smallest number to the user. Finally, you end the function:

```
:  return smaller
:endfunction
```

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
```

For people who like short functions, this does the same thing:

```
:function Min(num1, num2)
:  if a:num1 < a:num2
:    return a:num1
:  endif
:  return a:num2
: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)
```

Only now will the function be executed and the lines be interpreted by Vim. If there are mistakes, like using an undefined variable or function, you will now get an error message. When defining the function these errors are not detected.

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.

For example:

```
:function Show(start, ...)
:  echohl Title
:  echo "start is " . a:start
:  echohl None
:  let index = 1
:  while index <= a:0
:    echo "  Arg " . index . " is " . a:{index}
:    let index = index + 1
:  endwhile
:  echo ""
:endfunction
```

This uses the "`:echohl`" command to specify the highlighting used for the following "`:echo`" command. "`:echohl None`" stops it again. The "`:echon`" command works like "`:echo`", but doesn't output a line break.

You can also use the a:000 variable, it is a List of all the "`...`" arguments. See |`:h a:000`|.

Listing functions

The ":function" command lists the names and arguments of all user-defined functions:

```
:function
   function Show(start, ...)
   function GetVimIndent()
   function SetSyn(name)
```

To see what a function does, use its name as an argument for ":function":

```
:function SetSyn
   1     if &syntax == ''
   2       let &syntax = a:name
   3     endif
      endfunction
```

Debugging

The line number is useful for when you get an error message or when debugging. See |:h 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
```

You get an error when the function doesn't exist.

Function references

Sometimes it can be useful to have a variable point to one function or another. You can do it with the function() 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.

Lists and Dictionaries

So far we have used the basic types String and Number. Vim also supports two composite types: List and Dictionary.

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
```

This will loop over each element in List "`alist`", assigning the value to variable "`n`". The generic form of a for loop is:

```
:for {varname} in {listexpression}
:  {commands}
:endfor
```

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
```

Notice that the first item of the List that `range()` produces is zero, thus the last item is one less than the length of the list.

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>, ...}
```

An empty Dictionary is one without any keys:

```
{}
```

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

Now that you can put both values and functions in a Dictionary, 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 |`:h list-identity`| and |`:h 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 |`:h Lists`| and |`:h Dictionaries`|.

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: |`:h exception-handling`|.

Various remarks

Here is a summary of items that apply to Vim scripts. They are also mentioned elsewhere, but form a nice checklist.

The end-of-line character depends on the system. For Unix a single <NL> character is used. For MS-DOS, Windows, OS/2 and the like, <CR><LF> is used. This is important when using mappings that end in a <CR>. See |`:h :source_crnl`|.

White space

Blank lines are allowed and ignored.

Leading whitespace characters (blanks and TABs) are always ignored. The whitespaces between parameters (e.g. between the `'set'` and the `'cpoptions'` in the example below) are reduced to one blank character and plays the role of a separator, the whitespaces after the last (visible) character may or may not be ignored depending on the situation, see below.

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
```

The same example written as:

```
:set tags=my nice file
```

will issue an error, because it is interpreted as:

```
:set tags=my
:set nice
:set file
```

Comments

The character `"` (the double quote mark) starts a comment. Everything after and including this character until the end-of-line is considered a comment and is ignored, except for commands that don't consider comments, as shown in examples below. A comment can start on any character position on the line.

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: |`:h :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. |:h :#!| |:h -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
```

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.

In this section the first type is explained. Most items are also relevant for writing filetype plugins. The specifics for filetype plugins are in the next section |`write-filetype-plugin`|.

Name

First of all you must choose a name for your plugin. The features provided by the plugin should be clear from its name. And it should be unlikely that someone else writes a plugin with the same name but which does something different. And please limit the name to 8 characters, to avoid problems on old Windows systems.

A script that corrects typing mistakes could be called "`typecorr.vim`". We will use it here as an example.

For the plugin to work for everybody, it should follow a few guidelines. This will be explained step-by-step. The complete example plugin is at the end.

Body

Let's start with the body of the plugin, the lines that do the actual work:

```
14  iabbrev teh the
15  iabbrev otehr other
16  iabbrev wnat want
17  iabbrev synchronisation
18      \ synchronization
19  let s:count = 4
```

The actual list should be much longer, of course.

The line numbers have only been added to explain a few things, don't put them in your plugin file!

Header

You will probably add new corrections to the plugin and soon have several versions laying around. And when distributing this file, people will want to know who wrote this wonderful plugin and where they can send remarks. Therefore, put a header at the top of your plugin:

```
1  " Vim global plugin for correcting typing mistakes
2  " Last Change:  2000 Oct 15
3  " Maintainer:   Bram Moolenaar <Bram@vim.org>
```

About copyright and licensing: Since plugins are very useful and it's hardly worth restricting their distribution, please consider making your plugin either public domain or use the Vim |`:h license`|. A short note about this near the top of the plugin should be sufficient. Example:

```
4  " License:  This file is placed in the public domain.
```

Line continuation, avoiding side effects

In line 18 above, the line-continuation mechanism is used |`:h line-continuation`|. Users with `'compatible'` set will run into trouble here, they will get an error message. We can't just reset `'compatible'`, because that has a lot of side effects. To avoid this, we will set the `'cpoptions'` option to its Vim default value and restore it later. That will allow the use of line-continuation and make the script work for most people. It is done like this:

```
11  let s:save_cpo = &cpo
12  set cpo&vim
..
42  let &cpo = s:save_cpo
```

We first store the old value of `'cpoptions'` in the `s:save_cpo` variable. At the end of the plugin this value is restored.

Notice that a script-local variable is used |`:h s:var`|. A global variable could already be in use for something else. Always use script-local variables for things that are only used in the script.

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:

```
6  if exists("g:loaded_typecorr")
7    finish
8  endif
9  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:

```
22    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. |`:h :map-<unique>`|

But what if the user wants to define his own key sequence? We can allow that with this mechanism:

```
21  if !hasmapto('<Plug>TypecorrAdd')
22    map <unique> <Leader>a  <Plug>TypecorrAdd
23  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:

```
30  function s:Add(from, correct)
31    let to = input("type the correction for " . a:from . ": ")
32    exe ":iabbrev " . a:from . " " . to
..
36  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:

```
24  noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
..
28  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. |:h :map-<script>| The same is done in line 26 for ":noremenu". |:h :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 the difference 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. |:h :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. |:h 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:

```
 1  " Vim global plugin for correcting typing mistakes
 2  " Last Change:  2000 Oct 15
 3  " Maintainer:   Bram Moolenaar <Bram@vim.org>
 4  " License:  This file is placed in the public domain.
 5
 6  if exists("g:loaded_typecorr")
 7    finish
 8  endif
 9  let g:loaded_typecorr = 1
10
11  let s:save_cpo = &cpo
12  set cpo&vim
13
14  iabbrev teh the
15  iabbrev otehr other
16  iabbrev wnat want
17  iabbrev synchronisation
18      \ synchronization
19  let s:count = 4
20
21  if !hasmapto('<Plug>TypecorrAdd')
22    map <unique> <Leader>a  <Plug>TypecorrAdd
23  endif
24  noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
25
26  noremenu <script> Plugin.Add\ Correction      <SID>Add
27
28  noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
29
30  function s:Add(from, correct)
31    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 |`:h :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 |:h :source_crnl|. To be sure it is set right, do this before writing the file:

```
:set fileformat=unix
```

Documentation

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.

Here is a simple example for a plugin help file, called "typecorr.txt":

```
 1  *typecorr.txt*  Plugin for correcting typing mistakes
 2
 3  If you make typing mistakes, this plugin will have them corrected
 4  automatically.
 5
 6  There are currently only a few corrections.  Add your own if you like.
 7
 8  Mappings:
 9  <Leader>a   or   <Plug>TypecorrAdd
10      Add a correction for the word under the cursor.
11
12  Commands:
13  :Correct {word}
14      Add a correction for {word}.
15
16                                          *typecorr-settings*
17  This plugin doesn't have any settings.
```

The first line is actually the only one for which the format matters. It will be extracted from the help file to be put in the "LOCAL ADDITIONS:" section of help.txt |:h local-additions|. The first "*" must be in the first column of the first line. After adding your help file do ":help" and check that the entries line up nicely.

You can add more tags inside ** in your help file. But be careful not to use existing help tags. You would probably use the name of your plugin in most of them, like "typecorr-settings" in the example.

Using references to other parts of the help in || is recommended. This makes it easy for the user to find associated help.

Filetype detection

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 |:h new-filetype|.

Summary

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.

Writing a filetype plugin

A filetype plugin is like a global plugin, except that it sets options and defines mappings for the current buffer only. See |add-filetype-plugin| for how this type of plugin is used.

First read the section on global plugins above |Writing a plugin|. All that is said there also applies to filetype plugins. There are a few extras, which are explained here. The essential thing is that a filetype plugin should only have an effect on the current buffer.

Disabling

If you are writing a filetype plugin to be used by many people, they need a chance to disable loading it. Put this at the top of the plugin:

```
" Only do this when not done yet for this buffer
if exists("b:did_ftplugin")
  finish
endif
let b:did_ftplugin = 1
```

This also needs to be used to avoid that the same plugin is executed twice for the same buffer (happens when using an ":edit" command without arguments).

Now users can disable loading the default plugin completely by making a filetype plugin with only this line:

```
let b:did_ftplugin = 1
```

This does require that the filetype plugin directory comes before $VIMRUNTIME in 'runtimepath'!

If you do want to use the default plugin, but overrule one of the settings, you can write the different setting in a script:

```
setlocal textwidth=70
```

Now write this in the "after" directory, so that it gets sourced after the distributed "vim.vim" ftplugin |:h after-directory|. For Unix this would be "~/.vim/after/ftplugin/vim.vim". Note that the default plugin will have set "b:did_ftplugin", but it is ignored here.

Options

To make sure the filetype plugin only affects the current buffer use the

```
:setlocal
```

command to set options. And only set options which are local to a buffer (see the help for the option to check that). When using |:h :setlocal| for global options or options local to a window, the value will change for many buffers, and that is not what a filetype plugin should do.

When an option has a value that is a list of flags or items, consider using "+=" and "-=" to keep the existing value. Be aware that the user may have changed an option value already. First resetting to the default value and then changing it is often a good idea. Example:

```
:setlocal formatoptions& formatoptions+=ro
```

Mappings

To make sure mappings will only work in the current buffer use the

```
:map <buffer>
```

command. This needs to be combined with the two-step mapping explained above. An example of how to define functionality in a filetype plugin:

```
if !hasmapto('<Plug>JavaImport')
  map <buffer> <unique> <LocalLeader>i <Plug>JavaImport
endif
noremap <buffer> <unique> <Plug>JavaImport oimport ""<Left><Esc>
```

|:h hasmapto()| is used to check if the user has already defined a map to <Plug>JavaImport. If not, then the filetype plugin defines the default mapping. This starts with |:h <LocalLeader>|, which allows the user to select the key(s) he wants filetype plugin mappings to start with. The default is a backslash. "<unique>" is used to give an error message if the mapping already exists or overlaps with an existing mapping. |:h :noremap| is used to avoid that any other mappings that the user has defined interferes. You might want to use ":noremap <script>" to allow remapping mappings defined in this script that start with <SID>.

The user must have a chance to disable the mappings in a filetype plugin, without disabling everything. Here is an example of how this is done for a plugin for the mail filetype:

```
" Add mappings, unless the user didn't want this.
if !exists("no_plugin_maps") && !exists("no_mail_maps")
  " Quote text by inserting "> "
  if !hasmapto('<Plug>MailQuote')
    vmap <buffer> <LocalLeader>q <Plug>MailQuote
    nmap <buffer> <LocalLeader>q <Plug>MailQuote
  endif
  vnoremap <buffer> <Plug>MailQuote :s/^/> /<CR>
  nnoremap <buffer> <Plug>MailQuote :.,$s/^/> /<CR>
endif
```

Two global variables are used:

`no_plugin_maps`	disables mappings for all filetype plugins
`no_mail_maps`	disables mappings for a specific filetype

User commands

To add a user command for a specific file type, so that it can only be used in one buffer, use the "`-buffer`" argument to |`:h :command`|. Example:

```
:command -buffer  Make  make %:r.s
```

Variables

A filetype plugin will be sourced for each buffer of the type it's for. Local script variables |`:h s:var`| will be shared between all invocations. Use local buffer variables |`:h b:var`| if you want a variable specifically for one buffer.

Functions

When defining a function, this only needs to be done once. But the filetype plugin will be sourced every time a file with this filetype will be opened. This construct makes sure the function is only defined once:

```
:if !exists("*s:Func")
:  function s:Func(arg)
:    ...
:  endfunction
:endif
```

Undo

When the user does "`:setfiletype xyz`" the effect of the previous filetype should be undone. Set the `b:undo_ftplugin` variable to the commands that will undo the settings in your filetype plugin. Example:

```
let b:undo_ftplugin = "setlocal fo< com< tw< commentstring<"
    \ . "| unlet b:match_ignorecase b:match_words b:match_skip"
```

Using "`:setlocal`" with "`<`" after the option name resets the option to its global value. That is mostly the best way to reset the option value.

This does require removing the "`C`" flag from `'cpoptions'` to allow line continuation, as mentioned above |`use-cpo-save`|.

File name

The filetype must be included in the file name |`ftplugin-name`|. Use one of these three forms:

```
.../ftplugin/stuff.vim
.../ftplugin/stuff_foo.vim
.../ftplugin/stuff/bar.vim
```

"`stuff`" is the filetype, "`foo`" and "`bar`" are arbitrary names.

Summary

Summary of special things to use in a filetype plugin:

`<LocalLeader>`	Value of "`maplocalleader`", which the user defines as the keys that filetype plugin mappings start
`:map <buffer>`	Define a mapping local to the buffer.
`:noremap <script>`	Only remap mappings defined in this script that start with <SID>.
`:setlocal`	Set an option for the current buffer only.
`:command -buffer`	Define a user command local to the buffer.
`exists("*s:Func")`	Check if a function was already defined.

Also see |`plugin-special`|, the special things used for all plugins.

Writing a compiler plugin

A compiler plugin sets options for use with a specific compiler. The user can load it with the |`:h :compiler`| command. The main use is to set the `'errorformat'` and `'makeprg'` options.

Easiest is to have a look at examples. This command will edit all the default compiler plugins:

```
:next $VIMRUNTIME/compiler/*.vim
```

Use |`:h :next`| to go to the next plugin file.

There are two special items about these files. First is a mechanism to allow a user to overrule or add to the default file. The default files start with:

```
:if exists("current_compiler")
:  finish
:endif
:let current_compiler = "mine"
```

When you write a compiler file and put it in your personal runtime directory (e.g., `~/.vim/compiler` for Unix), you set the "`current_compiler`" variable to make the default file skip the settings. The second mechanism is to use "`:set`" for "`:compiler!`" and "`:setlocal`" for "`:compiler`". Vim defines the "`:CompilerSet`" user command for this. However, older Vim versions don't, thus your plugin should define it then. This is an example:

```
if exists(":CompilerSet") != 2
  command -nargs=* CompilerSet setlocal <args>
endif
CompilerSet errorformat&       " use the default 'errorformat'
CompilerSet makeprg=nmake
```

When you write a compiler plugin for the Vim distribution or for a system-wide runtime directory, use the mechanism mentioned above. When "`current_compiler`" was already set by a user plugin nothing will be done.

When you write a compiler plugin to overrule settings from a default plugin, don't check "`current_compiler`". This plugin is supposed to be loaded last, thus it should be in a directory at the end of `'runtimepath'`. For Unix that could be `~/.vim/after/compiler`.

Writing a plugin that loads quickly

A plugin may grow and become quite long. The startup delay may become noticeable, while you hardly ever use the plugin. Then it's time for a quickload plugin.

The basic idea is that the plugin is loaded twice. The first time user commands and mappings are defined that offer the functionality. The second time the functions that implement the functionality are defined.

It may sound surprising that quickload means loading a script twice. What we mean is that it loads quickly the first time, postponing the bulk of the script to the second time, which only happens when you actually use it. When you always use the functionality it actually gets slower!

Note that since Vim 7 there is an alternative: use the |`:h autoload`| functionality |`Writing library scripts`|.

The following example shows how it's done:

```
" Vim global plugin for demonstrating quick loading
" Last Change:  2005 Feb 25
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" License:  This file is placed in the public domain.

if !exists("s:did_load")
    command -nargs=* BNRead  call BufNetRead(<f-args>)
    map <F19> :call BufNetWrite('something')<CR>

    let s:did_load = 1
    exe 'au FuncUndefined BufNet* source ' . expand('<sfile>')
    finish
endif

function BufNetRead(...)
    echo 'BufNetRead(' . string(a:000) . ')'
    " read functionality here
endfunction

function BufNetWrite(...)
    echo 'BufNetWrite(' . string(a:000) . ')'
    " write functionality here
endfunction
```

When the script is first loaded "`s:did_load`" is not set. The commands between the "`if`" and "`endif`" will be executed. This ends in a |`:h :finish`| command, thus the rest of the script is not executed.

The second time the script is loaded "`s:did_load`" exists and the commands after the "`endif`" are executed. This defines the (possible long) BufNetRead() and BufNetWrite() functions.

If you drop this script in your plugin directory Vim will execute it on startup. This is the sequence of events that happens:

1. The "`BNRead`" command is defined and the <F19> key is mapped when the script is sourced at startup. A |`:h FuncUndefined`| autocommand is defined. The "`:finish`" command causes the script to terminate early.
2. The user types the BNRead command or presses the <F19> key. The `BufNetRead()` or `BufNetWrite()` function will be called.

3. Vim can't find the function and triggers the |`:h FuncUndefined`| autocommand event. Since the pattern "`BufNet*`" matches the invoked function, the command "`source fname`" will be executed. "`fname`" will be equal to the name of the script, no matter where it is located, because it comes from expanding "`<sfile>`" (see |`:h expand()`|).
4. The script is sourced again, the "`s:did_load`" variable exists and the functions are defined.

Notice that the functions that are loaded afterwards match the pattern in the |`:h FuncUndefined`| autocommand. You must make sure that no other plugin defines functions that match this pattern.

Writing library scripts

Some functionality will be required in several places. When this becomes more than a few lines you will want to put it in one script and use it from many scripts. We will call that one script a library script.

Manually loading a library script is possible, so long as you avoid loading it when it's already done. You can do this with the |`:h exists()`| function. Example:

```
if !exists('*MyLibFunction')
   runtime library/mylibscript.vim
endif
call MyLibFunction(arg)
```

Here you need to know that `MyLibFunction()` is defined in a script "`library/mylibscript.vim`" in one of the directories in `'runtimepath'`.

To make this a bit simpler Vim offers the autoload mechanism. Then the example looks like this:

```
call mylib#myfunction(arg)
```

That's a lot simpler, isn't it? Vim will recognize the function name and when it's not defined search for the script "`autoload/mylib.vim`" in `'runtimepath'`. That script must define the "`mylib#myfunction()`" function.

You can put many other functions in the mylib.vim script, you are free to organize your functions in library scripts. But you must use function names where the part before the '`#`' matches the script name. Otherwise Vim would not know what script to load.

If you get really enthusiastic and write lots of library scripts, you may want to use subdirectories. Example:

```
call netlib#ftp#read('somefile')
```

For Unix the library script used for this could be:

```
~/.vim/autoload/netlib/ftp.vim
```

Where the function is defined like this:

```
function netlib#ftp#read(fname)
    "  Read the file fname through ftp
endfunction
```

Notice that the name the function is defined with is exactly the same as the name used for calling the function. And the part before the last '`#`' exactly matches the subdirectory and script name.

You can use the same mechanism for variables:

```
let weekdays = dutch#weekdays
```

This will load the script "`autoload/dutch.vim`", which should contain something like:

```
let dutch#weekdays = ['zondag', 'maandag', 'dinsdag', 'woensdag',
    \ 'donderdag', 'vrijdag', 'zaterdag']
```

Further reading: |`:h autoload`|.

Distributing Vim scripts

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 |`:h vimball`|.

It’s good if you add a line to allow automatic updating. See |`:h glvs-plugins`|.

42. 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.

Introduction

The menus that Vim uses are defined in the file "`$VIMRUNTIME/menu.vim`". If you want to write your own menus, you might first want to look through that file.

To define a menu item, use the "`:menu`" command. The basic form of this command is as follows:

```
:menu {menu-item} {keys}
```

The `menu-item` describes where on the menu to put the item. A typical `menu-item` is`File.Save`", which represents the item "`Save`" under the "`File` "menu. A dot is used to separate the names. Example:

```
:menu File.Save  :update<CR>
```

The "`:update`" command writes the file when it was modified.

You can add another level: "`Edit.Settings.Shiftwidth`" defines a submenu "`Settings`" under the "`Edit`" menu, with an item "`Shiftwidth`". You could use even deeper levels. Don't use this too much, you need to move the mouse quite a bit to use such an item.

The " "`:menu`" " command is very similar to the "`:map`" command: the left side specifies how the item is triggered and the right hand side defines the characters that are executed. `keys` are characters, they are used just like you would have typed them. Thus in Insert mode, when `keys` is plain text, that text is inserted.

Accelerators

The ampersand character (`&`) is used to indicate an accelerator. For instance, you can use Alt-F to select "`File`" and S to select "`Save`". (The `'winaltkeys'` option may disable this though!). Therefore, the `menu-item` looks like "`&File.&Save`". The accelerator characters will be underlined in the menu.

You must take care that each key is used only once in each menu. Otherwise you will not know which of the two will actually be used. Vim doesn't warn you for this.

Priorities

The actual definition of the `File.Save` menu item is as follows:

```
:menu 10.340 &File.&Save<Tab>:w  :confirm w<CR>
```

The number 10.340 is called the priority number. It is used by the editor to decide where it places the menu item. The first number (10) indicates the position on the menu bar. Lower numbered menus are positioned to the left, higher numbers to the right.

These are the priorities used for the standard menus:

```
   10    20     40     50      60       70       9999

+-------------------------------------------------------------+
| File Edit  Tools  Syntax  Buffers  Window      Help         |
+-------------------------------------------------------------+
```

Notice that the Help menu is given a very high number, to make it appear on the far right.

The second number (340) determines the location of the item within the pull-down menu. Lower numbers go on top, higher number on the bottom. These are the priorities in the File menu:

```
        +-----------------+
10.310  |Open...          |
10.320  |Split-Open...    |
10.325  |New              |
10.330  |Close            |
10.335  |---------------- |
10.340  |Save             |
10.350  |Save As...       |
10.400  |---------------- |
10.410  |Split Diff with  |
10.420  |Split Patched By |
10.500  |---------------- |
10.510  |Print            |
10.600  |---------------- |
10.610  |Save-Exit        |
10.620  |Exit             |
        +-----------------+
```

Notice that there is room in between the numbers. This is where you can insert your own items, if you really want to (it's often better to leave the standard menus alone and add a new menu for your own items).

When you create a submenu, you can add another "`.number`" to the priority. Thus each name in `menu-item` has its priority number.

Special characters

The `menu-item` in this example is "`&File.&Save<Tab>:w`". This brings up an important point: `menu-item` must be one word. If you want to put a dot, space or tabs in the name, you either use the <> notation (<Space> and <Tab>, for instance) or use the backslash (
) escape.

```
:menu 10.305 &File.&Do\ It\.\.\. :exit<CR>
```

In this example, the name of the menu item "`Do It...`" contains a space and the command is "`:exit<CR>`".

The <Tab> character in a menu name is used to separate the part that defines the menu name from the part that gives a hint to the user. The part after the <Tab> is displayed right aligned in the menu. In the File.Save menu the name used is "`&File.&Save<Tab>:w`". Thus the menu name is "`File.Save`" and the hint is "`:w`".

Separators

The separator lines, used to group related menu items together, can be defined by using a name that starts and ends in a '`-`'. For example "`-sep-`". When using several separators the names must be different. Otherwise the names don't matter.

The command from a separator will never be executed, but you have to define one anyway. A single colon will do. Example:

```
:amenu 20.510 Edit.-sep3- :
```

Menu commands

You can define menu items that exist for only certain modes. This works just like the variations on the ":map" command:

:menu	Normal, Visual and Operator-pending mode
:nmenu	Normal mode
:vmenu	Visual mode
:omenu	Operator-pending mode
:menu!	Insert and Command-line mode
:imenu	Insert mode
:cmenu	Command-line mode
:amenu	All modes

To avoid that the commands of a menu item are being mapped, use the command ":noremenu", ":nnoremenu", ":anoremenu", etc.

Using :amenu

The ":amenu" command is a bit different. It assumes that the keys you give are to be executed in Normal mode. When Vim is in Visual or Insert mode when the menu is used, Vim first has to go back to Normal mode. ":amenu" inserts a CTRL-C or CTRL-O for you. For example, if you use this command:

```
:amenu  90.100 Mine.Find\ Word  *
```

Then the resulting menu commands will be:

Normal mode:	*
Visual mode:	CTRL-C *
Operator-pending mode:	CTRL-C *
Insert mode:	CTRL-O *
Command-line mode:	CTRL-C *

When in Command-line mode the CTRL-C will abandon the command typed so far. In Visual and Operator-pending mode CTRL-C will stop the mode. The CTRL-O in Insert mode will execute the command and then return to Insert mode.

CTRL-O only works for one command. If you need to use two or more commands, put them in a function and call that function. Example:

```
:amenu  Mine.Next\ File  :call <SID>NextFile()<CR>
:function <SID>NextFile()
:  next
:  1/^Code
:endfunction
```

This menu entry goes to the next file in the argument list with ":next". Then it searches for the line that starts with "Code".

The <SID> before the function name is the script ID. This makes the function local to the current Vim script file. This avoids problems when a function with the same name is defined in another script file. See |:h <SID>|.

Silent menus

The menu executes the `keys` as if you typed them. For a ":" command this means you will see the command being echoed on the command line. If it's a long command, the hit-Enter prompt will appear. That can be very annoying!

To avoid this, make the menu silent. This is done with the <silent> argument. For example, take the call to NextFile() in the previous example. When you use this menu, you will see this on the command line:

```
:call <SNR>34_NextFile()
```

To avoid this text on the command line, insert "`<silent>`" as the first argument:

```
:amenu <silent> Mine.Next\ File :call <SID>NextFile()<CR>
```

Don't use "`<silent>`" too often. It is not needed for short commands. If you make a menu for someone else, being able the see the executed command will give him a hint about what he could have typed, instead of using the mouse.

Listing menus

When a menu command is used without a `keys` part, it lists the already defined menus. You can specify a `menu-item,` or part of it, to list specific menus. Example:

```
:amenu
```

This lists all menus. That's a long list! Better specify the name of a menu to get a shorter list:

```
:amenu Edit
```

This lists only the "`Edit`" menu items for all modes. To list only one specific menu item for Insert mode:

```
:imenu Edit.Undo
```

Take care that you type exactly the right name. Case matters here. But the '`&`' for accelerators can be omitted. The <Tab> and what comes after it can be left out as well.

Deleting menus

To delete a menu, the same command is used as for listing, but with "`menu`" changed to "`unmenu`". Thus "`:menu`" becomes, "`:unmenu`", "`:nmenu`" becomes "`:nunmenu`", etc. To delete the "`Tools.Make`" item for Insert mode:

```
:iunmenu Tools.Make
```

You can delete a whole menu, with all its items, by using the menu name. Example:

```
:aunmenu Syntax
```

This deletes the Syntax menu and all the items in it.

Various

You can change the appearance of the menus with flags in `'guioptions'`. In the default value they are all included, except "`M`". You can remove a flag with a command like:

```
:set guioptions-=m
```

m	When removed the menubar is not displayed.
M	When added the default menus are not loaded.
g	When removed the inactive menu items are not made grey but are completely removed. (Does not work on all systems.)
t	When removed the tearoff feature is not enabled.

The dotted line at the top of a menu is not a separator line. When you select this item, the menu is "teared-off": It is displayed in a separate window. This is called a tearoff menu. This is useful when you use the same menu often.

For translating menu items, see |`:h :menutrans`|.

Since the mouse has to be used to select a menu item, it is a good idea to use the "`:browse`" command for selecting a file. And "`:confirm`" to get a dialog instead of an error message, e.g., when the current buffer contains changes. These two can be combined:

```
:amenu File.Open  :browse confirm edit<CR>
```

The "`:browse`" makes a file browser appear to select the file to edit. The "`:confirm`" will pop up a dialog when the current buffer has changes. You can then select to save the changes, throw them away or cancel the command.

For more complicated items, the `confirm()` and `inputdialog()` functions can be used. The default menus contain a few examples.

Toolbar and popup menus

There are two special menus: ToolBar and PopUp. Items that start with these names do not appear in the normal menu bar.

Toolbar

The toolbar appears only when the "`T`" flag is included in the `'guioptions'` option.

The toolbar uses icons rather than text to represent the command. For example, the `menu-item` named "`ToolBar.New`" causes the "`New`" icon to appear on the toolbar.

The Vim editor has 28 built-in icons. You can find a table here: |`:h builtin-tools`|. Most of them are used in the default toolbar. You can redefine what these items do (after the default menus are setup).

You can add another bitmap for a toolbar item. Or define a new toolbar item with a bitmap. For example, define a new toolbar item with:

```
:tmenu ToolBar.Compile  Compile the current file
:amenu ToolBar.Compile  :!cc % -o %:r<CR>
```

Now you need to create the icon. For MS-Windows it must be in bitmap format, with the name "`Compile.bmp`". For Unix XPM format is used, the file name is "`Compile.xpm`". The size must be 18 by 18 pixels. On MS-Windows other sizes can be used as well, but it will look ugly.

Put the bitmap in the directory "`bitmaps`" in one of the directories from `'runtimepath'`. E.g., for Unix " `/.vim/bitmaps/Compile.xpm`".

You can define tooltips for the items in the toolbar. A tooltip is a short text that explains what a toolbar item will do. For example "`Open file`". It appears when the mouse pointer is on the item, without moving for a moment. This is very useful if the meaning of the picture isn't that obvious. Example:

```
:tmenu ToolBar.Make  Run make in the current directory
```

Note: Pay attention to the case used. "`Toolbar`" and "`toolbar`" are different from "`ToolBar`"

To remove a tooltip, use the |`:h :tunmenu`| command.

The `'toolbar'` option can be used to display text instead of a bitmap, or both text and a bitmap. Most people use just the bitmap, since the text takes quite a bit of space.

Popup menu

The popup menu pops up where the mouse pointer is. On MS-Windows you activate it by clicking the right mouse button. Then you can select an item with the left mouse button. On Unix the popup menu is used by pressing and holding the right mouse button.

The popup menu only appears when the `'mousemodel'` has been set to "`popup`" or "`popup_setpos`". The difference between the two is that "`popup_setpos`" moves the cursor to the mouse pointer position. When clicking inside a selection, the selection will be used unmodified. When there is a selection but you click outside of it, the selection is removed.

There is a separate popup menu for each mode. Thus there are never grey items like in the normal menus.

What is the meaning of life, the universe and everything? Douglas Adams, the only person who knew what this question really was about is now dead, unfortunately. So now you might wonder what the meaning of death is...

43. 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.

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:

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 |:h 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 |:h <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|.

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 |Autocommands|.) 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 |:h 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.

44. 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 |`:h :syn-define`| in the reference manual.

Basic syntax commands

Using an existing syntax file to start with will save you a lot of time. Try finding a syntax file in `$VIMRUNTIME/syntax` for a language that is similar. These files will also show you the normal layout of a syntax file. To understand it, you need to read the following.

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
```

You can use this to check which items have actually been defined. Quite useful when you are experimenting with a new syntax file. It also shows the colors used for each item, which helps to find out what is what.

To list the items in a specific syntax group use:

```
:syntax list {group-name}
```

This also can be used to list clusters (explained in |`Clusters`|). Just include the `@` in the 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.

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
```

This example uses the group names "`xType`" and "`xStatement`". By convention, each group name is prefixed by the filetype for the language being defined. This example defines syntax for the x language (eXample language without an interesting name). In a syntax file for "`csh`" scripts the name "`cshType`" would be used. Thus the prefix is equal to the value of `'filetype'`.

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 |`:h 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.

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 |`:h pattern`| for help on search patterns.

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.

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 |:h :syn-contains| for the details.

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.

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: |`:h :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.

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
```

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.

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: |`:h :syn-sync`|.

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 |`Adding a filetype`|.

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
```

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.

45. 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.

Language for Messages

When you start Vim, it checks the environment to find out what language you are using. Mostly this should work fine, and you get the messages in your language (if they are available). To see what the current language is, use this command:

```
:language
```

If it replies with "`C`", this means the default is being used, which is English.

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.

What if you would like your messages in a different language? There are several ways. Which one you should use depends on the capabilities of your system.

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
```

This only works if the language is available on your system. The advantage is that all the GUI messages and things in libraries will use the right language as well. A disadvantage is that you must do this before starting Vim. If you want to change language while Vim is running, you can use the second method:

```
:language fr_FR.ISO_8859-1
```

This way you can try out several names for your language. You will get an error message when it's not supported on your system. You don't get an error when translated messages are not available. Vim will silently fall back to using English.

To find out which languages are supported on your system, find the directory where they are listed. On my system it is "`/usr/share/locale`". On some systems it's in "`/usr/lib/locale`". The manual page for "`setlocale`" should give you a hint where it is found on your system.

Be careful to type the name exactly as it should be. Upper and lowercase matter, and the '`-`' and '`_`' characters are easily confused.

You can also set the language separately for messages, edited text and the time format. See |`:h :language`|.

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.

Language for Menus

The default menus are in English. To be able to use your local language, they must be translated. Normally this is automatically done for you if the environment is set for your language, just like with messages. You don't need to do anything extra for this. But it only works if translations for the language are available.

Suppose you are in Germany, with the language set to German, but prefer to use "`File`" instead of "`Datei`". You can switch back to using the English menus this way:

```
:set langmenu=none
```

It is also possible to specify a language:

```
:set langmenu=nl_NL.ISO_8859-1
```

Like above, differences between "`-`" and "`_`" matter. However, upper/lowercase differences are ignored here.

The `'langmenu'` option must be set before the menus are loaded. Once the menus have been defined changing `'langmenu'` has no direct effect. Therefore, put the command to set `'langmenu'` in your vimrc file.

If you really want to switch menu language while running Vim, you can do it this way:

```
:source $VIMRUNTIME/delmenu.vim
:set langmenu=de_DE.ISO_8859-1
:source $VIMRUNTIME/menu.vim
```

There is one drawback: All menus that you defined yourself will be gone. You will need to redefine them as well.

Do-it-yourself menu translation

To see which menu translations are available, look in this directory:

```
$VIMRUNTIME/lang
```

The files are called `menu_{language}.vim`. If you don't see the language you want to use, you can do your own translations. The simplest way to do this is by copying one of the existing language files, and change it.

First find out the name of your language with the "`:language`" command. Use this name, but with all letters made lowercase. Then copy the file to your own runtime directory, as found early in `'runtimepath'`. For example, for Unix you would do:

```
:!cp $VIMRUNTIME/lang/menu_ko_kr.euckr.vim ~/.vim/lang/menu_nl_be.iso_8859-1.vim
```

You will find hints for the translation in "`$VIMRUNTIME/lang/README.txt`".

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

The nice thing about Unicode is that other encodings can be converted to it and back without losing information. When you make Vim use Unicode internally, you will be able to edit files in any encoding.

Unfortunately, the number of systems supporting Unicode is still limited. Thus it's unlikely that your language uses it. You need to tell Vim you want to use Unicode, and how to handle interfacing with the rest of the system.

Let's start with the GUI version of Vim, which is able to display Unicode characters. This should work:

```
:set encoding=utf-8
:set guifont=-misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

The `'encoding'` option tells Vim the encoding of the characters that you use. This applies to the text in buffers (files you are editing), registers, Vim script files, etc. You can regard `'encoding'` as the setting for the internals of Vim.

This example assumes you have this font on your system. The name in the example is for the X Window System. This font is in a package that is used to enhance xterm with Unicode support. If you don't have this font, you might find it here: http://www.cl.cam.ac.uk/~mgk25/download/ucs-fonts.tar.gz.

For MS-Windows, some fonts have a limited number of Unicode characters. Try using the "Courier New" font. You can use the Edit/Select Font... menu to select and try out the fonts available. Only fixed-width fonts can be used though. Example:

```
:set guifont=courier_new:h12
```

If it doesn't work well, try getting a fontpack. If Microsoft didn't move it, you can find it here: http://www.microsoft.com/typography/fonts/default.aspx

Now you have told Vim to use Unicode internally and display text with a Unicode font. Typed characters still arrive in the encoding of your original language. This requires converting them to Unicode. Tell Vim the language from which to convert with the `'termencoding'` option. You can do it like this:

```
:let &termencoding = &encoding
:set encoding=utf-8
```

This assigns the old value of `'encoding'` to `'termencoding'` before setting `'encoding'` to utf-8. You will have to try out if this really works for your setup. It should work especially well when using an input method for an Asian language, and you want to edit Unicode text.

Using unicode in a unicode terminal

There are terminals that support Unicode directly. The standard xterm that comes with XFree86 is one of them. Let's use that as an example.

First of all, the xterm must have been compiled with Unicode support. See |`:h UTF8-xterm`| how to check that and how to compile it when needed.

Start the xterm with the "`-u8`" argument. You might also need so specify a font. Example:

```
xterm -u8 -fn -misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

Now you can run Vim inside this terminal. Set `'encoding'` to "`utf-8`" as before. That's all.

Using unicode in an ordinary terminal

Suppose you want to work with Unicode files, but don't have a terminal with Unicode support. You can do this with Vim, although characters that are not supported by the terminal will not be displayed. The layout of the text will be preserved.

```
:let &termencoding = &encoding
:set encoding=utf-8
```

This is the same as what was used for the GUI. But it works differently: Vim will convert the displayed text before sending it to the terminal. That avoids that the display is messed up with strange characters.

For this to work the conversion between `'termencoding'` and `'encoding'` must be possible. Vim will convert from latin1 to Unicode, thus that always works. For other conversions the |`:h +iconv`| feature is required.

Try editing a file with Unicode characters in it. You will notice that Vim will put a question mark (or underscore or some other character) in places where a character should be that the terminal can't display. Move the cursor to a question mark and use this command:

```
ga
```

Vim will display a line with the code of the character. This gives you a hint about what character it is. You can look it up in a Unicode table. You could actually view a file that way, if you have lots of time at hand.

Note: Since `'encoding'` is used for all text inside Vim, changing it makes all non-ASCII text invalid. You will notice this when using registers and the `'viminfo'` file (e.g., a remembered search pattern). It's recommended to set `'encoding'` in your vimrc file, and leave it alone.

Editing files with a different encoding

Suppose you have setup Vim to use Unicode, and you want to edit a file that is in 16-bit Unicode. Sounds simple, right? Well, Vim actually uses utf-8 encoding internally, thus the 16-bit encoding must be converted, since there is a difference between the character set (Unicode) and the encoding (utf-8 or 16-bit).

Vim will try to detect what kind of file you are editing. It uses the encoding names in the `'fileencodings'` option. When using Unicode, the default value is: "`ucs-bom,utf-8,latin1`". This means that Vim checks the file to see if it's one of these encodings:

`ucs-bom`	File must start with a Byte Order Mark (BOM). This allows detection of 16-bit, 32-bit and utf-8 Unicode encodings.
`utf-8`	utf-8 Unicode. This is rejected when a sequence of bytes is illegal in utf-8.
`latin1`	The good old 8-bit encoding. Always works.

When you start editing that 16-bit Unicode file, and it has a BOM, Vim will detect this and convert the file to utf-8 when reading it. The `'fileencoding'` option (without s at the end) is set to the detected value. In this case it is "`utf-16le`". That means it's Unicode, 16-bit and little-endian. This file format is common on MS-Windows (e.g., for registry files).

When writing the file, Vim will compare 'fileencoding' with 'encoding'. If they are different, the text will be converted.

An empty value for 'fileencoding' means that no conversion is to be done. Thus the text is assumed to be encoded with 'encoding'.

If the default 'fileencodings' value is not good for you, set it to the encodings you want Vim to try. Only when a value is found to be invalid will the next one be used. Putting "latin1" first doesn't work, because it is never illegal. An example, to fall back to Japanese when the file doesn't have a BOM and isn't utf-8:

```
:set fileencodings=ucs-bom,utf-8,sjis
```

See |:h encoding-values| for suggested values. Other values may work as well. This depends on the conversion available.

Forcing an encoding

If the automatic detection doesn't work you must tell Vim what encoding the file is. Example:

```
:edit ++enc=koi8-r russian.txt
```

The "++enc" part specifies the name of the encoding to be used for this file only. Vim will convert the file from the specified encoding, Russian in this example, to 'encoding'. 'fileencoding' will also be set to the specified encoding, so that the reverse conversion can be done when writing the file.

The same argument can be used when writing the file. This way you can actually use Vim to convert a file. Example:

```
:write ++enc=utf-8 russian.txt
```

Note: Conversion may result in lost characters. Conversion from an encoding to Unicode and back is mostly free of this problem, unless there are illegal characters. Conversion from Unicode to other encodings often loses information when there was more than one language in the file.

Entering language text

Computer keyboards don't have much more than a hundred keys. Some languages have thousands of characters, Unicode has ten thousands. So how do you type these characters?

First of all, when you don't use too many of the special characters, you can use digraphs. This was already explained in |Digraphs|.

When you use a language that uses many more characters than keys on your keyboard, you will want to use an Input Method (IM). This requires learning the translation from typed keys to resulting character. When you need an IM you probably already have one on your system. It should work with Vim like with other programs. For details see |:h mbyte-XIM| for the X Window system and |:h mbyte-IME| for MS-Windows.

Keymaps

For some languages the character set is different from latin, but uses a similar number of characters. It's possible to map keys to characters. Vim uses keymaps for this.

Suppose you want to type Hebrew. You can load the keymap like this:

```
:set keymap=hebrew
```

Vim will try to find a keymap file for you. This depends on the value of 'encoding'. If no matching file was found, you will get an error message.

Now you can type Hebrew in Insert mode. In Normal mode, and when typing a ":" command, Vim automatically switches to English. You can use this command to switch between Hebrew and English:

```
CTRL-^
```

This only works in Insert mode and Command-line mode. In Normal mode it does something completely different (jumps to alternate file).

The usage of the keymap is indicated in the mode message, if you have the 'showmode' option set. In the GUI Vim will indicate the usage of keymaps with a different cursor color.

You can also change the usage of the keymap with the 'iminsert' and 'imsearch' options.

To see the list of mappings, use this command:

```
:lmap
```

To find out which keymap files are available, in the GUI you can use the Edit/Keymap menu. Otherwise you can use this command:

```
:echo globpath(&rtp, "keymap/*.vim")
```

Do-it-yourself keymaps

You can create your own keymap file. It's not very difficult. Start with a keymap file that is similar to the language you want to use. Copy it to the "keymap" directory in your runtime directory. For example, for Unix, you would use the directory "~/.vim/keymap".

The name of the keymap file must look like this:

```
keymap/{name}.vim
```

or

```
keymap/{name}_{encoding}.vim
```

{name} is the name of the keymap. Chose a name that is obvious, but different from existing keymaps (unless you want to replace an existing keymap file). {name} cannot contain an underscore. Optionally, add the encoding used after an underscore. Examples:

```
keymap/hebrew.vim
keymap/hebrew_utf-8.vim
```

The contents of the file should be self-explanatory. Look at a few of the keymaps that are distributed with Vim. For the details, see |:h mbyte-keymap|.

Last resort

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

Don't type the spaces. See |`:h i_CTRL-V_digit`| for the details.

90. Installing Vim

Before you can use Vim you have to install it. Depending on your system it's simple or easy. This chapter gives a few hints and also explains how upgrading to a new version is done.

Unix

First you have to decide if you are going to install Vim system-wide or for a single user. The installation is almost the same, but the directory where Vim is installed in differs.

For a system-wide installation the base directory "`/usr/local`" is often used. But this may be different for your system. Try finding out where other packages are installed.

When installing for a single user, you can use your home directory as the base. The files will be placed in subdirectories like "`bin`" and "`shared/vim`".

From a package

You can get precompiled binaries for many different UNIX systems. There is a long list with links on this page: http://www.vim.org/binaries.html

Volunteers maintain the binaries, so they are often out of date. It is a good idea to compile your own UNIX version from the source. Also, creating the editor from the source allows you to control which features are compiled. This does require a compiler though.

If you have a Linux distribution, the "`vi`" program is probably a minimal version of Vim. It doesn't do syntax highlighting, for example. Try finding another Vim package in your distribution, or search on the web site.

From sources

To compile and install Vim, you will need the following:

- A C compiler (GCC preferred)
- The GZIP program (you can get it from www.gnu.org)
- The Vim source and runtime archives

To get the Vim archives, look in this file for a mirror near you, this should provide the fastest download:

ftp://ftp.vim.org/pub/vim/MIRRORS

Or use the home site ftp.vim.org, if you think it's fast enough. Go to the "`unix`" directory and you'll find a list of files there. The version number is embedded in the file name. You will want to get the most recent version.

You can get the files for Unix in two ways: One big archive that contains everything, or four smaller ones that each fit on a floppy disk. For version 6.1 the single big one is called:

```
vim-6.1.tar.bz2
```

You need the bzip2 program to uncompress it. If you don't have it, get the four smaller files, which can be uncompressed with gzip. For Vim 6.1 they are called:

```
vim-6.1-src1.tar.gz
vim-6.1-src2.tar.gz
vim-6.1-rt1.tar.gz
vim-6.1-rt2.tar.gz
```

Compiling

First create a top directory to work in, for example:

```
mkdir ~/vim
cd ~/vim
```

Then unpack the archives there. If you have the one big archive, you unpack it like this:

```
bzip2 -d -c path/vim-6.1.tar.bz2 | tar xf -
```

Change "`path`" to where you have downloaded the file.

```
gzip -d -c path/vim-6.1-src1.tar.gz | tar xf -
gzip -d -c path/vim-6.1-src2.tar.gz | tar xf -
gzip -d -c path/vim-6.1-rt1.tar.gz | tar xf -
gzip -d -c path/vim-6.1-rt2.tar.gz | tar xf -
```

If you are satisfied with getting the default features, and your environment is setup properly, you should be able to compile Vim with just this:

```
cd vim61/src
make
```

The make program will run configure and compile everything. Further on we will explain how to compile with different features.

If there are errors while compiling, carefully look at the error messages. There should be a hint about what went wrong. Hopefully you will be able to correct it. You might have to disable some features to make Vim compile. Look in the Makefile for specific hints for your system.

Testing

Now you can check if compiling worked OK:

```
make test
```

This will run a sequence of test scripts to verify that Vim works as expected. Vim will be started many times and all kinds of text and messages flash by. If it is alright you will finally see:

```
test results:
ALL DONE
```

If you get "`TEST FAILURE`" some test failed. If there are one or two messages about failed tests, Vim might still work, but not perfectly. If you see a lot of error messages or Vim doesn't finish until the end, there must be something wrong. Either try to find out yourself, or find someone who can solve it. You could look in the |`maillist-archive`| for a solution. If everything else fails, you could ask in the vim |`maillist`| if someone can help you.

Installing

If you want to install in your home directory, edit the Makefile and search for a line:

```
#prefix = $(HOME)
```

Remove the `#` at the start of the line.

When installing for the whole system, Vim has most likely already selected a good installation directory for you. You can also specify one, see below. You need to become root for the following.

To install Vim do:

```
make install
```

That should move all the relevant files to the right place. Now you can try running vim to verify that it works. Use two simple tests to check if Vim can find its runtime files:

```
:help
:syntax enable
```

If this doesn't work, use this command to check where Vim is looking for the runtime files:

```
:echo $VIMRUNTIME
```

You can also start Vim with the "`-V`" argument to see what happens during startup:

```
vim -V
```

Don't forget that the user manual assumes you Vim in a certain way. After installing Vim, follow the instructions at |`not-compatible`| to make Vim work as assumed in this manual.

Selecting features

Vim has many ways to select features. One of the simple ways is to edit the Makefile. There are many directions and examples. Often you can enable or disable a feature by uncommenting a line.

An alternative is to run "`configure`" separately. This allows you to specify configuration options manually. The disadvantage is that you have to figure out what exactly to type.

Some of the most interesting configure arguments follow. These can also be enabled from the Makefile.

`--prefix=directory`	Top directory where to install Vim.
`--with-features=tiny`	Compile with many features disabled.
`--with-features=small`	Compile with some features disabled.
`--with-features=big`	Compile with more features enabled.
`--with-features=huge`	Compile with most features enabled. See |`:h +feature-list`| for which feature is enabled in which case.
`--enable-perlinterp`	Enable the Perl interface. There are similar arguments for ruby, python and tcl.
`--disable-gui`	Do not compile the GUI interface.
`--without-x`	Do not compile X-windows features. When both of these are used, Vim will not connect to the X server, which makes startup faster.

To see the whole list use:

```
./configure --help
```

You can find a bit of explanation for each feature, and links for more information here: |`:h feature-list`|.

For the adventurous, edit the file "`feature.h`". You can also change the source code yourself!

MS-Windows

There are two ways to install the Vim program for Microsoft Windows. You can uncompress several archives, or use a self-installing big archive. Most users with fairly recent computers will prefer the second method. For the first one, you will need:

- An archive with binaries for Vim.
- The Vim runtime archive.
- A program to unpack the zip files.

To get the Vim archives, look in this file for a mirror near you, this should provide the fastest download:

```
ftp://ftp.vim.org/pub/vim/MIRRORS
```

Or use the home site ftp.vim.org, if you think it's fast enough. Go to the "`pc`" directory and you'll find a list of files there. The version number is embedded in the file name. You will want to get the most recent version. We will use "`61`" here, which is version 6.1.

```
gvim61.exe      The self-installing archive.
```

This is all you need for the second method. Just launch the executable, and follow the prompts.

For the first method you must chose one of the binary archives. These are available:

`gvim61.zip`	The normal MS-Windows GUI version.
`gvim61ole.zip`	The MS-Windows GUI version with OLE support. Uses more memory, supports interfacing with other OLE applications.
`vim61w32.zip`	32 bit MS-Windows console version. For use in a Win NT/2000/XP console. Does not work well on Win 95/98.
`vim61d32.zip`	32 bit MS-DOS version. For use in the Win 95/98 console window.
`vim61d16.zip`	16 bit MS-DOS version. Only for old systems. Does not support long filenames.

You only need one of them. Although you could install both a GUI and a console version. You always need to get the archive with runtime files.

```
vim61rt.zip     The runtime files.
```

Use your un-zip program to unpack the files. For example, using the "`unzip`" program:

```
cd c:\
unzip path\gvim61.zip
unzip path\vim61rt.zip
```

This will unpack the files in the directory "`c:\vim\vim61`". If you already have a "`vim`" directory somewhere, you will want to move to the directory just above it.

Now change to the "`vim\vim61`" directory and run the install program:

```
install
```

Carefully look through the messages and select the options you want to use. If you finally select "`do it`" the install program will carry out the actions you selected.

The install program doesn't move the runtime files. They remain where you unpacked them.

In case you are not satisfied with the features included in the supplied binaries, you could try compiling Vim yourself. Get the source archive from the same location as where the binaries are. You need a compiler for which a makefile exists. Microsoft Visual C works, but is expensive. The Free Borland command-line compiler 5.5 can be used, as well as the free MingW and Cygwin compilers. Check the file src/INSTALLpc.txt for hints.

Upgrading

If you are running one version of Vim and want to install another, here is what to do.

Unix

When you type "`make install`" the runtime files will be copied to a directory which is specific for this version. Thus they will not overwrite a previous version. This makes it possible to use two or more versions next to each other.

The executable "`vim`" will overwrite an older version. If you don't care about keeping the old version, running "`make install`" will work fine. You can delete the old runtime files manually. Just delete the directory with the version number in it and all files below it. Example:

```
rm -rf /usr/local/share/vim/vim58
```

There are normally no changed files below this directory. If you did change the "`filetype.vim`" file, for example, you better merge the changes into the new version before deleting it.

If you are careful and want to try out the new version for a while before switching to it, install the new version under another name. You need to specify a configure argument. For example:

```
./configure --with-vim-name=vim6
```

Before running "`make install`", you could use "`make -n install`" to check that no valuable existing files are overwritten.

When you finally decide to switch to the new version, all you need to do is to rename the binary to "`vim`". For example:

```
mv /usr/local/bin/vim6 /usr/local/bin/vim
```

Ms-windows

Upgrading is mostly equal to installing a new version. Just unpack the files in the same place as the previous version. A new directory will be created, e.g., "`vim61`", for the files of the new version. Your runtime files, vimrc file, viminfo, etc. will be left alone.

If you want to run the new version next to the old one, you will have to do some handwork. Don't run the install program, it will overwrite a few files of the old version. Execute the new binaries by specifying the full path. The program should be able to automatically find the runtime files for the right version. However, this won't work if you set the `$VIMRUNTIME` variable somewhere.

If you are satisfied with the upgrade, you can delete the files of the previous version. See |`Uninstalling Vim`|.

Common installation issues

This section describes some of the common problems that occur when installing Vim and suggests some solutions. It also contains answers to many installation questions.

Q: I Do Not Have Root Privileges. How Do I Install Vim? (Unix)

Use the following configuration command to install Vim in a directory called `$HOME/vim`:

```
./configure --prefix=$HOME
```

This gives you a personal copy of Vim. You need to put `$HOME/bin` in your path to execute the editor. Also see |`install-home`|.

Q: The Colors Are Not Right on My Screen. (Unix)

Check your terminal settings by using the following command in a shell:

```
echo $TERM
```

If the terminal type listed is not correct, fix it. For more hints, see |`No or wrong colors?`|. Another solution is to always use the GUI version of Vim, called gvim. This avoids the need for a correct terminal setup.

Q: My Backspace And Delete Keys Don't Work Right

The definition of what key sends what code is very unclear for backspace <BS> and Delete <Del> keys. First of all, check your `$TERM` setting. If there is nothing wrong with it, try this:

```
:set t_kb=^V<BS>
:set t_kD=^V<Del>
```

In the first line you need to press CTRL-V and then hit the backspace key. In the second line you need to press CTRL-V and then hit the Delete key. You can put these lines in your vimrc file, see |`The vimrc file`|. A disadvantage is that it won't work when you use another terminal some day. Look here for alternate solutions: |`:h :fixdel`|.

Q: I Am Using RedHat Linux. Can I Use the Vim That Comes with the System?

By default RedHat installs a minimal version of Vim. Check your RPM packages for something named "`Vim-enhanced-version.rpm`" and install that.

Q: How Do I Turn Syntax Coloring On? How do I make plugins work?

Use the example vimrc script. You can find an explanation on how to use it here: |`not-compatible`|.

See chapter 6 for information about syntax highlighting: |`Using syntax highlighting`|.

Q: What Is a Good vimrc File to Use?

See the www.vim.org Web site for several good examples.

Q: Where Do I Find a Good Vim Plugin?

See the Vim-online site: http://vim.sf.net. Many users have uploaded useful Vim scripts and plugins there.

Q: Where Do I Find More Tips? See the Vim-online site: http://vim.sf.net. There is an archive with hints from Vim users. You might also want to search in the |`maillist-archive`|.

Uninstalling Vim

In the unlikely event you want to uninstall Vim completely, this is how you do it.

Unix

When you installed Vim as a package, check your package manager to find out how to remove the package again.

If you installed Vim from sources you can use this command:

```
make uninstall
```

However, if you have deleted the original files or you used an archive that someone supplied, you can't do this. Do delete the files manually, here is an example for when "`/usr/local`" was used as the root:

```
rm -rf /usr/local/share/vim/vim61
rm /usr/local/bin/eview
rm /usr/local/bin/evim
rm /usr/local/bin/ex
rm /usr/local/bin/gview
rm /usr/local/bin/gvim
rm /usr/local/bin/gvim
rm /usr/local/bin/gvimdiff
rm /usr/local/bin/rgview
rm /usr/local/bin/rgvim
rm /usr/local/bin/rview
rm /usr/local/bin/rvim
rm /usr/local/bin/rvim
rm /usr/local/bin/view
rm /usr/local/bin/vim
rm /usr/local/bin/vimdiff
rm /usr/local/bin/vimtutor
rm /usr/local/bin/xxd
rm /usr/local/man/man1/eview.1
rm /usr/local/man/man1/evim.1
rm /usr/local/man/man1/ex.1
rm /usr/local/man/man1/gview.1
rm /usr/local/man/man1/gvim.1
rm /usr/local/man/man1/gvimdiff.1
rm /usr/local/man/man1/rgview.1
rm /usr/local/man/man1/rgvim.1
rm /usr/local/man/man1/rview.1
rm /usr/local/man/man1/rvim.1
rm /usr/local/man/man1/view.1
rm /usr/local/man/man1/vim.1
rm /usr/local/man/man1/vimdiff.1
rm /usr/local/man/man1/vimtutor.1
rm /usr/local/man/man1/xxd.1
```

Ms-windows

If you installed Vim with the self-installing archive you can run the "`uninstall-gui`" program located in the same directory as the other Vim programs, e.g. "`c:\vim\vim61`". You can also launch it from the Start menu if installed the Vim entries there. This will remove most of the files, menu entries and desktop shortcuts. Some files may remain however, as they need a Windows restart before being deleted.

You will be given the option to remove the whole "`vim`" directory. It probably contains your vimrc file and other runtime files that you created, so be careful.

Else, if you installed Vim with the zip archives, the preferred way is to use the "`uninstal`" program (note the missing l at the end). You can find it in the same directory as the "`install`" program, e.g., "`c:\vim\vim61`". This should also work from the usual "install/remove software" page.

However, this only removes the registry entries for Vim. You have to delete the files yourself. Simply select the directory "`vim\vim61`" and delete it recursively. There should be no files there that you changed, but you might want to check that first.

The "`vim`" directory probably contains your vimrc file and other runtime files that you created. You might want to keep that.