Getting Started
|usr_01.txt| About the manuals
|usr_02.txt| The first steps in Vim
|usr_03.txt| Moving around
|usr_04.txt| Making small changes
|usr_05.txt| Set your settings
|usr_06.txt| Using syntax highlighting
|usr_07.txt| Editing more than one file
|usr_08.txt| Splitting windows
|usr_09.txt| Using the GUI
|usr_10.txt| Making big changes
|usr_11.txt| Recovering from a crash
|usr_12.txt| Clever tricks

==
usr_01.txt   For Vim version 8.0. Last change: 2017 Jul 15

                    VIM USER MANUAL - by Bram Moolenaar

                            About the manuals

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

|01.1| Two manuals
|01.2| Vim installed
|01.3| Using the Vim tutor
|01.4| Copyright

     Next chapter: |usr_02.txt| The first steps in Vim
Table of contents: |usr_toc.txt|

==
01.1 Two manuals

The Vim documentation consists of two parts:

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

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

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

JUMPING AROUND

The text contains hyperlinks between the two parts, allowing you to quickly
jump between the description of an editing task and a precise explanation of
the commands and options used for it. Use these two commands:

        Press CTRL-] to jump to a subject under the cursor.
        Press CTRL-O to jump back (repeat to go further back).

Many links are in vertical bars, like this: |bars|. The bars themselves may
be hidden or invisible, see below. An option name, like 'number', a command
in double quotes like ":write" and any other word can also be used as a link.
Try it out: Move the cursor to CTRL-] and press CTRL-] on it.

Other subjects can be found with the ":help" command, see |help.txt|.

The bars and stars are usually hidden with the |conceal| feature. They also use |hl-Ignore|, using the same color for the text as the background. You can make them visible with: >
	:set conceallevel=0
	:hi link HelpBar Normal
	:hi link HelpStar Normal

==
01.2 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: |usr_90.txt|.
not-compatible
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 |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 |evim-keys| for details.

==
01.3 Using the Vim tutor *tutor* *vimtutor*

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 --clean -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 --clean -c 'e $VIMRUNTIME/tutor/tutor.fr' -c 'w! TUTORCOPY' -c 'q'
<
2. Edit the copied file with Vim:
>
        vim --clean TUTORCOPY
<
   The --clean argument makes sure Vim is started with nice defaults.

3. Delete the copied file when you are finished with it:
>
        del TUTORCOPY
<
==
01.4 Copyright   *manual-copyright*

frombook
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 |iccf|.

==

Next chapter: |usr_02.txt| The first steps in Vim

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_02.txt For Vim version 8.0. Last change: 2017 Mar 14

VIM USER MANUAL - by Bram Moolenaar

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.

Next chapter: |usr_03.txt| Moving around
Previous chapter: |usr_01.txt| About the manuals
Table of contents: |usr_toc.txt|

==
02.1 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.

==========

02.2 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.

==
02.3 Moving around

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

```
h   left                                                    *hjkl*
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)
```

==
02.4 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 ~

==
02.5 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 |32.2|.

==========

02.6 Other editing commands

Vim has a large number of commands to change the text. See |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.

==============================================================================
*02.7*  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 |07.4|.


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.

==============================================================================
*02.8*  Finding help

Everything you always wanted to know can be found in the Vim help files.
Don't be afraid to ask!

If you know what you are looking for, it is usually easier to search for it
using the help system, instead of using Google.  Because the subjects follow
```

```
a certain style guide.

Also the help has the advantage of belonging to your particular Vim version.
You won't see help for commands added later.  These would not work for you.

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, |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 |29.1| 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 below: |help-summary|.

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-summary*  >

1) Use Ctrl-D after typing a topic and let Vim show all available topics.
   Or press Tab to complete: >
        :help some<Tab>
<   More information on how to use the help: >
        :help helphelp

2) Follow the links in bars to related help.  You can go from the detailed
   help to the user documentation, which describes certain commands more from
   a user perspective and less detailed.  E.g. after: >
        :help pattern.txt
<   You can see the user guide topics |03.9| and |usr_27.txt| in the
   introduction.

3) Options are enclosed in single apostrophes.  To go to the help topic for the
   list option: >
        :help 'list'
<   If you only know you are looking for a certain option, you can also do: >
        :help options.txt
<   to open the help page which describes all option handling and then search
   using regular expressions, e.g. textwidth.
   Certain options have their own namespace, e.g.: >
        :help cpo-<letter>
<   for the corresponding flag of the 'cpoptions' settings, substitute <letter>
   by a specific flag, e.g.: >
        :help cpo-;
<   And for the guioption flags: >
        :help go-<letter>
```

```
4) Normal mode commands do not have a prefix. To go to the help page for the
   "gt" command: >
        :help gt

5) Insert mode commands start with i_.  Help for deleting a word: >
        :help i_CTRL-W

6) Visual mode commands start with v_.  Help for jumping to the other side of
   the Visual area: >
        :help v_o

7) Command line editing and arguments start with c_.  Help for using the
   command argument %: >
        :help c_%

8) Ex-commands always start with ":", so to go to the :s command help: >
        :help :s

9) Commands specifically for debugging start with ">".  To go to the help
   for the "cont" debug command: >
        :help >cont

10) Key combinations.  They usually start with a single letter indicating
    the mode for which they can be used.  E.g.: >
        :help i_CTRL-X
<    takes you to the family of Ctrl-X commands for insert mode which can be
    used to auto complete different things.  Note, that certain keys will
    always be written the same, e.g. Control will always be CTRL.
    For normal mode commands there is no prefix and the topic is available at
    :h CTRL-<Letter>. E.g.  >
        :help CTRL-W
<    In contrast >
        :help c_CTRL-R
<    will describe what the Ctrl-R does when entering commands in the Command
    line and >
        :help v_Ctrl-A
<    talks about incrementing numbers in visual mode and >
        :help g_CTRL-A
<    talks about the g<C-A> command (e.g. you have to press "g" then <Ctrl-A>).
    Here the "g" stand for the normal command "g" which always expects a second
    key before doing something similar to the commands starting with "z"

11) Regexp items always start with /.  So to get help for the "\+" quantifier
    in Vim regexes: >
        :help /\+
<    If you need to know everything about regular expressions, start reading
    at: >
        :help pattern.txt

12) Registers always start with "quote". To find out about the special ":"
    register: >
        :help quote:

13) Vim script is available at >
        :help eval.txt
<   Certain aspects of the language are available at :h expr-X where "X" is a
   single letter. E.g.  >
        :help expr-!
<   will take you to the topic describing the "!" (Not) operator for
   VimScript.
   Also important is >
```

```
        :help function-list
<   to find a short description of all functions available.  Help topics for
   Vim script functions always include the "()", so: >
        :help append()
<   talks about the append Vim script function rather than how to append text
   in the current buffer.

14) Mappings are talked about in the help page :h |map.txt|. Use >
        :help mapmode-i
<    to find out about the |:imap| command.  Also use :map-topic
    to find out about certain subtopics particular for mappings.  e.g: >
        :help :map-local
<    for buffer-local mappings or >
        :help map-bar
<    for how the '|' is handled in mappings.

15) Command definitions are talked about :h command-topic, so use >
        :help command-bar
<    to find out about the '!' argument for custom commands.

16) Window management commands always start with CTRL-W, so you find the
    corresponding help at :h CTRL-W_letter.  E.g. >
        :help CTRL-W_p
<    for moving the previous accessed window.  You can also access >
        :help windows.txt
<    and read your way through if you are looking for window handling
    commands.

17) Use |:helpgrep| to search in all help pages (and also of any installed
    plugins).  See |:helpgrep| for how to use it.
    To search for a topic: >
        :helpgrep topic
<    This takes you to the first match.  To go to the next one: >
        :cnext
<    All matches are available in the quickfix window which can be opened
    with: >
        :copen
<    Move around to the match you like and press Enter to jump to that help.

18) The user manual.  This describes help topics for beginners in a rather
    friendly way.  Start at |usr_toc.txt| to find the table of content (as you
    might have guessed): >
        :help usr_toc.txt
<    Skim over the contents to find interesting topics. The "Digraphs" and
    "Entering special characters" items are in chapter 24, so to go to that
    particular help page: >
        :help usr_24.txt
<    Also if you want to access a certain chapter in the help, the chapter
    number can be accessed directly like this: >
        :help 10.1
<    goes to chapter 10.1 in |usr_10.txt| and talks about recording macros.

19) Highlighting groups.  Always start with hl-groupname.  E.g. >
        :help hl-WarningMsg
<    talks about the WarningMsg highlighting group.

20) Syntax highlighting is namespaced to :syn-topic e.g. >
        :help :syn-conceal
<    talks about the conceal argument for the :syn command.

21) Quickfix commands usually start with :c while location list commands
    usually start with :l
```

```
22) Autocommand events can be found by their name: >
        :help BufWinLeave
<    To see all possible events: >
        :help autocommand-events

23) Command-line switches always start with "-".  So for the help of the -f
    command switch of Vim use: >
        :help -f

24) Optional features always start with "+".  To find out about the
    conceal feature use: >
        :help +conceal

25) Documentation for included filetype specific functionality is usually
    available  in the form ft-<filetype>-<functionality>.  So >
        :help ft-c-syntax
<    talks about the C syntax file and the option it provides.  Sometimes,
    additional sections for omni completion >
        :help ft-php-omni
<    or filetype plugins >
        :help ft-tex-plugin
<    are available.

26) Error and Warning codes can be looked up directly in the help.  So >
        :help E297
<    takes you exactly to the description of the swap error message and >
        :help W10
<    talks about the warning "Changing a readonly file".
    Sometimes however, those error codes are not described, but rather are
    listed at the Vim command that usually causes this.  So: >
        :help E128
<    takes you to the |:function| command


==============================================================================

Next chapter: |usr_03.txt|  Moving around

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_03.txt*    For Vim version 8.0.  Last change: 2017 Jul 21

                     VIM USER MANUAL - by Bram Moolenaar

                             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 |Q_lr|.

|03.1|  Word movement
|03.2|  Moving to the start or end of a line
|03.3|  Moving to a character
|03.4|  Matching a parenthesis
|03.5|  Moving to a specific line
|03.6|  Telling where you are
|03.7|  Scrolling around
|03.8|  Simple searches
|03.9|  Simple search patterns
|03.10| Using marks
```

Next chapter: |usr_04.txt| Making small changes
Previous chapter: |usr_02.txt| The first steps in Vim
Table of contents: |usr_toc.txt|

==
03.1 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. If you try this out in the help directly, 'iskeyword' needs to be reset for the examples to work: >

```
        :set iskeyword&
```

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.

==
03.2 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.

==
03.3 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.

==
03.4 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) ~
               ---+---------------->
                  %
```

==
03.5 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) ~
```

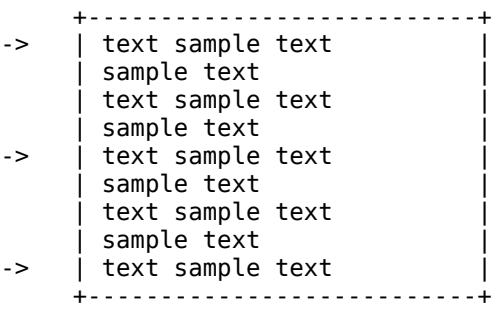
This tells you that you might want to fix something on line 33. So how do you
find line 33? One way is to do "9999k" to go to the top of the file and "32j"
to go down thirty-two lines. It is not a good way, but it works. A much
better way of doing things is to use the "G" command. With a count, this
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
|usr_30.txt|, 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:



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

==
03.6 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.

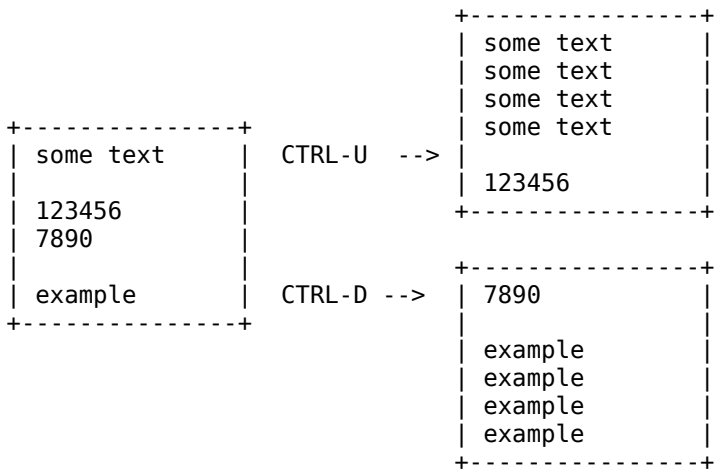
3. Set the 'ruler' option. This will display the cursor position in the
    lower right corner of the Vim window: >

        :set ruler

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

==
03.7 Scrolling around

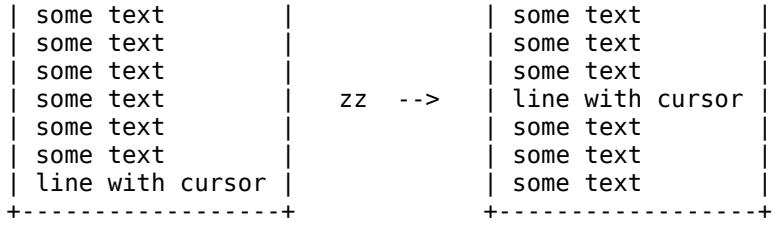
The CTRL-U command scrolls down half a screen of text. Think of looking through a viewing window at the text and moving this window up by half the height of the window. Thus the window moves up over the text, which is backward in the file. Don't worry if you have a little trouble remembering which end 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 |Q_sc|. To always keep a few lines of context around the cursor, use the 'scrolloff' option.

==
03.8 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 searches 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 the '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 beginning 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.

==
03.9 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 |usr_27.txt|. You can find the full
explanation here: |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.

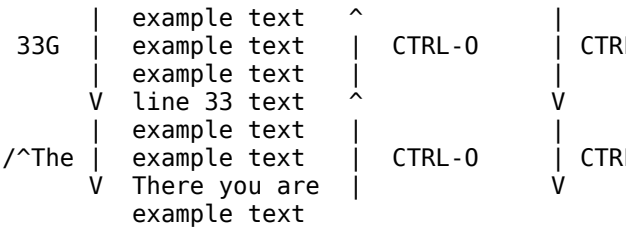
Let's mark the places where "/the" matches in this example line with "x"s:

        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 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.

==============================================================================
*03.10* 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.



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 *bookmark*

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

==============================================================================

Next chapter: |usr_04.txt|  Making small changes

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_04.txt*    For Vim version 8.0.  Last change: 2014 Aug 29

                     VIM USER MANUAL - by Bram Moolenaar

                             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.

|04.1|  Operators and motions
|04.2|  Changing text
|04.3|  Repeating a change
|04.4|  Visual mode
|04.5|  Moving text
|04.6|  Copying text
|04.7|  Using the clipboard
|04.8|  Text objects
|04.9|  Replace mode
|04.10| Conclusion

     Next chapter: |usr_05.txt|  Set your settings
 Previous chapter: |usr_03.txt|  Moving around
Table of contents: |usr_toc.txt|

==============================================================================
*04.1*  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.

==

04.2 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.

==
04.3 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.
```

==========

04.4 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.

==
04.5 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
```

==

04.6 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

==
04.7 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: |clipboard|.

==
04.8 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: |text-objects|.

==
04.9 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.

==
04.10 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: |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.

==============================================================================

Next chapter: |usr_05.txt|  Set your settings

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_05.txt*    For Vim version 8.0.  Last change: 2016 Mar 28

                     VIM USER MANUAL - by Bram Moolenaar

                             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.

|05.1|  The vimrc file
|05.2|  The example vimrc file explained
|05.3|  Simple mappings
|05.4|  Adding a package
|05.5|  Adding a plugin
|05.6|  Adding a help file
|05.7|  The option window
|05.8|  Often used options

     Next chapter: |usr_06.txt|  Using syntax highlighting
 Previous chapter: |usr_04.txt|  Making small changes
Table of contents: |usr_toc.txt|

==============================================================================
*05.1*  The vimrc file                                      *vimrc-intro*

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 |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 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: |usr_41.txt|.

==
05.2 The example vimrc file explained		*vimrc_example.vim*

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 |07.4|
>

        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 |usr_41.txt|.

vimrc-filetype >
```
        filetype plugin indent on
```

This switches on three very clever mechanisms:
1. Filetype detection.
   Whenever you start editing a file, Vim will try to figure out what kind of
   file this is. When you edit "main.c", Vim will see the ".c" extension and
   recognize this as a "c" filetype. When you edit a file that starts with
   "#!/bin/sh", Vim will recognize it as a "sh" filetype.
   The filetype detection is used for syntax highlighting and the other two
   items below.
   See |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 |:filetype-indent-on| and 'indentexpr'.

>
```
        autocmd FileType text setlocal textwidth=78
```

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

restore-cursor >
```
        autocmd BufReadPost *
            \ if line("'\"") > 1 && line("'\"") <= line("$") |
            \   exe "normal! g`\"" |
            \ endif
```

Another autocommand. This time it is used after reading any file. The
complicated stuff after it checks if the '" mark is defined, and jumps to it
if so. The backslash at the start of a line is used to continue the command
from the previous line. That avoids a line getting very long.

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

==
05.3 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 |40.1|.

==
05.4 Adding a package   *add-package* *matchit-install*

A package is a set of files that you can add to Vim. There are two kinds of
packages: optional and automatically loaded on startup.

The Vim distribution comes with a few packages that you can optionally use.
For example, the matchit plugin. 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).

To start using the matchit plugin, add one line to your vimrc file: >

```
        packadd! matchit

That's all!  After restarting Vim you can find help about this plugin: >
        :help matchit

This works, because when `:packadd` loaded the plugin it also added the
package directory in 'runtimepath', so that the help file can be found.

You can find packages on the Internet in various places.  It usually comes as
an archive or as a repository.  For an archive you can follow these steps:
        1. create the package directory: >
                mkdir -p ~/.vim/pack/fancy
<          "fancy" can be any name of your liking.  Use one that describes the
           package.
        2. unpack the archive in that directory.  This assumes the top
           directory in the archive is "start": >
                cd ~/.vim/pack/fancy
                unzip /tmp/fancy.zip
<          If the archive layout is different make sure that you end up with a
           path like this:
                ~/.vim/pack/fancy/start/fancytext/plugin/fancy.vim ~
           Here "fancytext" is the name of the package, it can be anything
           else.

More information about packages can be found here: |packages|.

==============================================================================
*05.5*  Adding a plugin                                  *add-plugin* *plugin*

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

There are two types of plugins:

    global plugin: Used for all kinds of files
  filetype plugin: Only used for a specific type of file

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


GLOBAL PLUGINS                                           *standard-plugin*

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

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


GETTING A GLOBAL PLUGIN

Where can you find plugins?
- Some come with Vim.  You can find them in the directory $VIMRUNTIME/macros
```

```
  and its sub-directories.
- Download from the net.  There is a large collection on http://www.vim.org.
- They are sometimes posted in a Vim |maillist|.
- You could write one yourself, see |write-plugin|.

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


USING A GLOBAL PLUGIN

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

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

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

        mkdir ~/.vim
        mkdir ~/.vim/plugin
        cp /tmp/yourplugin.vim ~/.vim/plugin

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

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                        *add-filetype-plugin* *ftplugins*

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                                    *ftplugin-name*

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:

|filetype-plugins|	Documentation for the filetype plugins and information about how to avoid that mappings cause problems.
|load-plugins|	When the global plugins are loaded during startup.
|ftplugin-overrule|	Overruling the settings from a global plugin.
|write-plugin|	How to write a plugin script.
|plugin-details|	For more information about using plugins or when your plugin doesn't work.
|new-filetype|	How to detect a new file type.

==
05.6 Adding a help file   *add-local-help*

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 "doit.vim" plugin as an example. This plugin comes with
documentation: "doit.txt". Let's first copy the plugin to the right
directory. This time we will do it from inside Vim. (You may skip some of

```
the "mkdir" commands if you already have the directory.) >

        :!mkdir ~/.vim
        :!mkdir ~/.vim/plugin
        :!cp /tmp/doit.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 /tmp/doit.txt ~/.vim/doc

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

        :helptags ~/.vim/doc

Now you can use the >

        :help doit

command to find help for "doit" 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|.

==============================================================================
*05.7*  The option window

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

        :options

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

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

        set wrap        nowrap ~

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

        set nowrap      wrap ~

The option has now been switched off.

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

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

        set so=0 ~

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

==
05.8 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.

==============================================================================

Next chapter: |usr_06.txt|  Using syntax highlighting

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_06.txt*    For Vim version 8.0.  Last change: 2009 Oct 28

                      VIM USER MANUAL - by Bram Moolenaar

                           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.

|06.1|  Switching it on
|06.2|  No or wrong colors?
|06.3|  Different colors
|06.4|  With colors or without colors
|06.5|  Printing with colors
|06.6|  Further reading

     Next chapter: |usr_07.txt|  Editing more than one file
 Previous chapter: |usr_05.txt|  Set your settings
Table of contents: |usr_toc.txt|

==============================================================================
*06.1*  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 |vimrc| file.

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

        if &t_Co > 1
           syntax enable
        endif

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

==============================================================================
*06.2*  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:
        |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 |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 |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 |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 |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 |:syn-sync|.  Some syntax files have a way to make
        it look further back, see the help for the specific syntax file.  For
        example, |tex.vim| for the TeX syntax.

==============================================================================
*06.3*  Different colors                                *:syn-default-override*

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 |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 |:highlight|
   command.

3. Tell Vim to always use your color scheme.  Put this line in your |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.

==============================================================================
*06.4*  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.

                                                        *:syn-off*
If you want to stop highlighting completely use: >

        :syntax off

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

                                                        *:syn-manual*
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
<
==============================================================================
*06.5*  Printing with colors                            *syntax-printing*

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 |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.

==
06.6 Further reading

|usr_44.txt| Your own syntax highlighted.
|syntax| All the details.

==

Next chapter: |usr_07.txt| Editing more than one file

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_07.txt   For Vim version 8.0. Last change: 2017 Sep 18

                     VIM USER MANUAL - by Bram Moolenaar

                          Editing 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.

|07.1| Edit another file
|07.2| A list of files
|07.3| Jumping from file to file
|07.4| Backup files
|07.5| Copy text between files
|07.6| Viewing a file
|07.7| Changing the file name

     Next chapter: |usr_08.txt| Splitting windows
 Previous chapter: |usr_06.txt| Using syntax highlighting

Table of contents: |usr_toc.txt|

==
07.1 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 |22.4|: The buffer list.

==
07.2 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?
arglist-quit

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).

==
07.3 Jumping from file to file

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

```
:args one.c two.c three.c
```

You are now in one.c. >

```
:next
```

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

```
:next
```

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

The file you were previously editing is called the "alternate" 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 through 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
down 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.

==
07.4 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.

==
07.5 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 placed 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.

==============================================================================
*07.6*  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

==============================================================================
*07.7*  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.

==============================================================================

Next chapter: |usr_08.txt|  Splitting windows

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_08.txt*    For Vim version 8.0.  Last change: 2017 Aug 11

                     VIM USER MANUAL - by Bram Moolenaar

                              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.

|08.1|  Split a window
|08.2|  Split a window on another file
|08.3|  Window size
|08.4|  Vertical splits
|08.5|  Moving windows
|08.6|  Commands for all windows
|08.7|  Viewing differences with vimdiff
|08.8|  Various
|08.9|  Tab pages

     Next chapter: |usr_09.txt|  Using the GUI
 Previous chapter: |usr_07.txt|  Editing more than one file
Table of contents: |usr_toc.txt|

==============================================================================
*08.1*  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 the 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.

==
08.2 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:

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

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.

==
08.3 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.

==
08.4 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: |Q_wi|.

==========

08.5 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:

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

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

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

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
```

==
08.6 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.

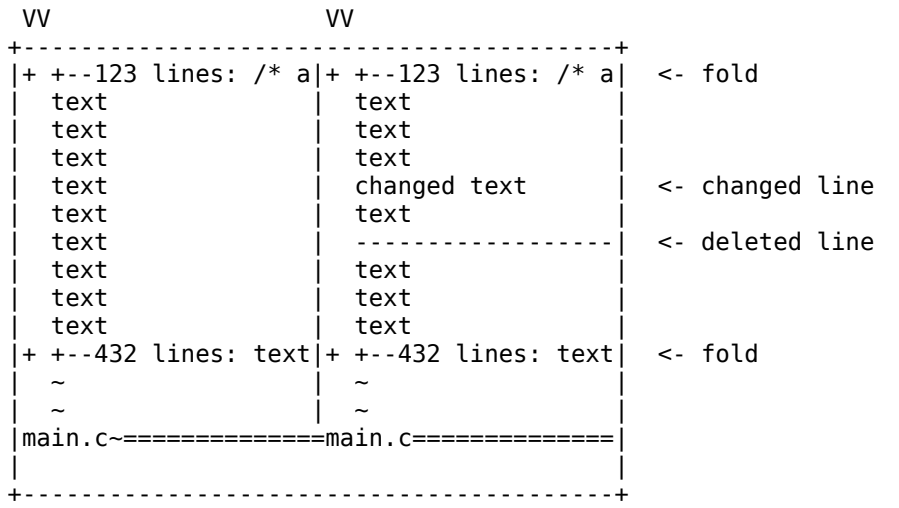
==
08.7 Viewing differences with vimdiff

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.



(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 |vimdiff|.

==
08.8 Various

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

        0   never
        1   only when there are split windows (the default)
        2   always

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

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

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

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

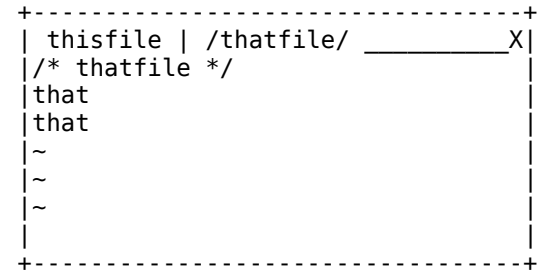
==
08.9 Tab pages

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:



(thatfile is bold)

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

each other, 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:

- click with the mouse in the space after the last label
        The next tab page will be selected, like with "gt".

- click with the mouse on the "X" in the top right corner
        The current tab page will be closed. Unless there are unsaved
        changes in the current tab page.

- double click with the mouse in the top line
        A new tab page will be created.

- the "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 |tab-page|.

==

Next chapter: |usr_09.txt| Using the GUI

usr_09.txt   For Vim version 8.0. Last change: 2017 Aug 11

VIM USER MANUAL - by Bram Moolenaar

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.

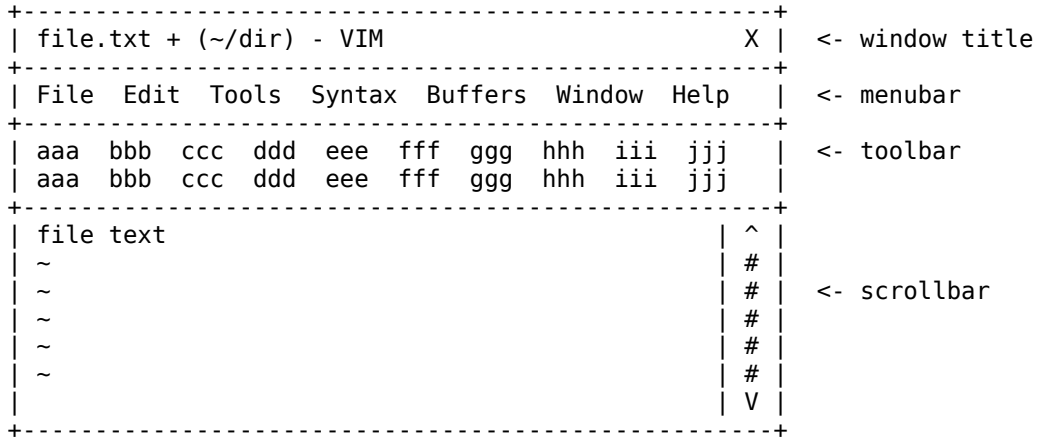
     Next chapter: |usr_10.txt| Making big changes
 Previous chapter: |usr_08.txt| Splitting windows
Table of contents: |usr_toc.txt|

==
09.1 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!).



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 characters 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'.

==============================================================================
*09.2*  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 |: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 |09.4|)
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

==============================================================================
*09.3*  The clipboard

In section |04.7| 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.

==
09.4 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.

==

Next chapter: |usr_10.txt| Making big changes

usr_10.txt For Vim version 8.0. Last change: 2006 Nov 05

VIM USER MANUAL - by Bram Moolenaar

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.

     Next chapter: |usr_11.txt| Recovering from a crash
 Previous chapter: |usr_09.txt| Using the GUI
Table of contents: |usr_toc.txt|

==
10.1 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.

==
10.2 Substitution *find-replace*

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 |10.3|.

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+
```

==========

10.3 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.

==
10.4 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 |: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.

==
10.5 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.

==============================================================================
*10.6*  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.

==============================================================================
*10.7*  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.

==============================================================================
*10.8*  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 ~

==============================================================================
*10.9*  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

==

Next chapter: |usr_11.txt| Recovering from a crash

usr_11.txt   For Vim version 8.0. Last change: 2010 Jul 20

VIM USER MANUAL - by Bram Moolenaar

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.

     Next chapter: |usr_12.txt| Clever tricks
 Previous chapter: |usr_10.txt| Making big changes
Table of contents: |usr_toc.txt|

==
11.1 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 |08.7|. 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.

==========

11.2 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.

==============================================================================
*11.3*  Crashed or not?                                  *ATTENTION* *E325*

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? *swap-exists-choices*

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 |:recover-crypt|.

==============================================================================
*11.4*  Further reading

|swap-file|     An explanation about where the swap file will be created and
               what its name is.
|:preserve|     Manually flushing the swap file to disk.
|: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.

==============================================================================

Next chapter: |usr_12.txt|  Clever tricks

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_12.txt*    For Vim version 8.0.  Last change: 2017 Aug 11
```

VIM USER MANUAL - by Bram Moolenaar

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.

Next chapter: |usr_20.txt| Typing command-line commands quickly
Previous chapter: |usr_11.txt| Recovering from a crash
Table of contents: |usr_toc.txt|

==========

12.1 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.

==============================================================================
*12.2*  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 |sub-replace-special|.

==
12.3 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.

==
12.4 Reverse line order

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

```
        :global/^/m 0
```

Abbreviated: >

```
        :g/^/m 0
```

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

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

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

==========

12.5 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 |count-items|.

==========

12.6 Find a man page *find-manpage*

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().

==
12.7 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, see |usr_27.txt|.
   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.

==
12.8 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'.

==

Next chapter: |usr_20.txt| Typing command-line commands quickly