Editing Effectively
|usr_20.txt| Typing command-line commands quickly
|usr_21.txt| Go away and come back
|usr_22.txt| Finding the file to edit
|usr_23.txt| Editing other files
|usr_24.txt| Inserting quickly
|usr_25.txt| Editing formatted text
|usr_26.txt| Repeating
|usr_27.txt| Search commands and patterns
|usr_28.txt| Folding
|usr_29.txt| Moving through programs
|usr_30.txt| Editing programs
|usr_31.txt| Exploiting the GUI
|usr_32.txt| The undo tree

==
usr_20.txt   For Vim version 8.0. Last change: 2006 Apr 24

                    VIM USER MANUAL - by Bram Moolenaar

                    Typing command-line commands quickly

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

|20.1| Command line editing
|20.2| Command line abbreviations
|20.3| Command line completion
|20.4| Command line history
|20.5| Command line window

     Next chapter: |usr_21.txt| Go away and come back
 Previous chapter: |usr_12.txt| Clever tricks
Table of contents: |usr_toc.txt|

==
20.1 Command line editing

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

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

        :s/col/pig/

Before you hit <Enter>, you notice that "col" should be "cow". To correct
this, you type <Left> five times. The cursor is now just after "col". Type
<BS> and "w" to correct: >

        :s/cow/pig/

Now you can press <Enter> directly. You don't have to move the cursor to the
end of the line before executing the command.

The most often used keys to move around in the command line:

```
        <Left>                  one character left
        <Right>                 one character right
        <S-Left> or <C-Left>    one word left
        <S-Right> or <C-Right>  one word right
        CTRL-B or <Home>        to begin of command line
        CTRL-E or <End>         to end of command line

        Note:
        <S-Left> (cursor left key with Shift key pressed) and <C-Left> (cursor
        left key with Control pressed) will not work on all keyboards.  Same
        for the other Shift and Control combinations.

You can also use the mouse to move the cursor.


DELETING

As mentioned, <BS> deletes the character before the cursor.  To delete a whole
word use CTRL-W.

        /the fine pig ~

                     CTRL-W

        /the fine ~

CTRL-U removes all text, thus allows you to start all over again.


OVERSTRIKE

The <Insert> key toggles between inserting characters and replacing the
existing ones.  Start with this text:

        /the fine pig ~

Move the cursor to the start of "fine" with <S-Left> twice (or <Left> eight
times, if <S-Left> doesn't work).  Now press <Insert> to switch to overstrike
and type "great":

        /the greatpig ~

Oops, we lost the space.  Now, don't use <BS>, because it would delete the
"t" (this is different from Replace mode).  Instead, press <Insert> to switch
from overstrike to inserting, and type the space:

        /the great pig ~


CANCELLING

You thought of executing a : or / command, but changed your mind.  To get rid
of what you already typed, without executing it, press CTRL-C or <Esc>.

        Note:
        <Esc> is the universal "get out" key.  Unfortunately, in the good old
        Vi pressing <Esc> in a command line executed the command!  Since that
        might be considered to be a bug, Vim uses <Esc> to cancel the command.
        But with the 'cpoptions' option it can be made Vi compatible.  And
        when using a mapping (which might be written for Vi) <Esc> also works
        Vi compatible.  Therefore, using CTRL-C is a method that always works.
```

If you are at the start of the command line, pressing <BS> will cancel the command. It's like deleting the ":" or "/" that the line starts with.

==
20.2 Command line abbreviations

Some of the ":" commands are really long. We already mentioned that ":substitute" can be abbreviated to ":s". This is a generic mechanism, all ":" commands can be abbreviated.

How short can a command get? There are 26 letters, and many more commands. For example, ":set" also starts with ":s", but ":s" doesn't start a ":set" command. Instead ":set" can be abbreviated to ":se".
When the shorter form of a command could be used for two commands, it stands for only one of them. There is no logic behind which one, you have to learn them. In the help files the shortest form that works is mentioned. For example: >

```
:s[ubstitute]
```

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

In the user manual we will either use the full name of command, or a short version that is still readable. For example, ":function" can be abbreviated to ":fu". But since most people don't understand what that stands for, we will use ":fun". (Vim doesn't have a ":funny" command, otherwise ":fun" would be confusing too.)

It is recommended that in Vim scripts you write the full command name. That makes it easier to read back when you make later changes. Except for some often used commands like ":w" (":write") and ":r" (":read").
A particularly confusing one is ":end", which could stand for ":endif", ":endwhile" or ":endfunction". Therefore, always use the full name.

SHORT OPTION NAMES

In the user manual the long version of the option names is used. Many options also have a short name. Unlike ":" commands, there is only one short name that works. For example, the short name of 'autoindent' is 'ai'. Thus these two commands do the same thing: >

```
:set autoindent
:set ai
```

You can find the full list of long and short names here: |option-list|.

==
20.3 Command line completion

This is one of those Vim features that, by itself, is a reason to switch from Vi to Vim. Once you have used this, you can't do without.

Suppose you have a directory that contains these files:

```
info.txt
intro.txt
bodyofthepaper.txt
```

To edit the last one, you use the command: >

```
        :edit bodyofthepaper.txt

It's easy to type this wrong.  A much quicker way is: >

        :edit b<Tab>

Which will result in the same command.  What happened?  The <Tab> key does
completion of the word before the cursor.  In this case "b".  Vim looks in the
directory and finds only one file that starts with a "b".  That must be the
one you are looking for, thus Vim completes the file name for you.

Now type: >

        :edit i<Tab>

Vim will beep, and give you: >

        :edit info.txt

The beep means that Vim has found more than one match.  It then uses the first
match it found (alphabetically).  If you press <Tab> again, you get: >

        :edit intro.txt

Thus, if the first <Tab> doesn't give you the file you were looking for, press
it again.  If there are more matches, you will see them all, one at a time.
   If you press <Tab> on the last matching entry, you will go back to what you
first typed: >

        :edit i

Then it starts all over again.  Thus Vim cycles through the list of matches.
Use CTRL-P to go through the list in the other direction:

              <------------------- <Tab> -------------------------+
                                                                  |
                  <Tab> -->                    <Tab> -->
        :edit i                :edit info.txt                :edit intro.txt
                  <-- CTRL-P                   <-- CTRL-P
           |
           +---------------------- CTRL-P ----------------------->


CONTEXT

When you type ":set i" instead of ":edit i" and press <Tab> you get: >

        :set icon

Hey, why didn't you get ":set info.txt"?  That's because Vim has context
sensitive completion.  The kind of words Vim will look for depends on the
command before it.  Vim knows that you cannot use a file name just after a
":set" command, but you can use an option name.
   Again, if you repeat typing the <Tab>, Vim will cycle through all matches.
There are quite a few, it's better to type more characters first: >

        :set isk<Tab>

Gives: >

        :set iskeyword
```

Now type "=" and press <Tab>: >

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

What happens here is that Vim inserts the old value of the option. Now you can edit it.
   What is completed with <Tab> is what Vim expects in that place. Just try it out to see how it works. In some situations you will not get what you want. That's either because Vim doesn't know what you want, or because completion was not implemented for that situation. In that case you will get a <Tab> inserted (displayed as ^I).

LIST MATCHES

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

```
:set is
```

results in: >

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

Vim lists the matches and then comes back with the text you typed. You can now check the list for the item you wanted. If it isn't there, you can use <BS> to correct the word. If there are many matches, type a few more characters before pressing <Tab> to complete the rest.
   If you have watched carefully, you will have noticed that "incsearch" doesn't start with "is". In this case "is" stands for the short name of "incsearch". (Many options have a short and a long name.) Vim is clever enough to know that you might have wanted to expand the short name of the option into the long name.

THERE IS MORE

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

The 'wildmode' option can be used to change the way completion works.
The 'wildmenu' option can be used to get a menu-like list of matches.
Use the 'suffixes' option to specify files that are less important and appear at the end of the list of files.
The 'wildignore' option specifies files that are not listed at all.

More about all of this here: |cmdline-completion|

==
20.4 Command line history

In chapter 3 we briefly mentioned the history. The basics are that you can use the <Up> key to recall an older command line. <Down> then takes you back to newer commands.

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

histories are for expressions and input lines for the input() function.
|cmdline-history|

Suppose you have done a ":set" command, typed ten more colon commands and then want to repeat that ":set" command again. You could press ":" and then ten times <Up>. There is a quicker way: >

```
:se<Up>
```

Vim will now go back to the previous command that started with "se". You have a good chance that this is the ":set" command you were looking for. At least you should not have to press <Up> very often (unless ":set" commands is all you have done).

The <Up> key will use the text typed so far and compare it with the lines in the history. Only matching lines will be used.
If you do not find the line you were looking for, use <Down> to go back to what you typed and correct that. Or use CTRL-U to start all over again.

To see all the lines in the history: >

```
:history
```

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

```
:history /
```

CTRL-P will work like <Up>, except that it doesn't matter what you already typed. Similarly for CTRL-N and <Down>. CTRL-P stands for previous, CTRL-N for next.

==========

20.5 Command line window

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

Open the command line window with this command: >

```
q:
```

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

```
+-------------------------------------+
|other window                         |
|~                                    |
|file.txt=============================|
|:e c                                 |
|:e config.h.in                       |
|:set path=.,/usr/include,,           |
|:set iskeyword=@,48-57,_,192-255     |
|:set is                              |
|:q                                   |
|:                                    |
|command-line=========================|
|                                     |
+-------------------------------------+
```

You are now in Normal mode. You can use the "hjkl" keys to move around. For
example, move up with "5k" to the ":e config.h.in" line. Type "$h" to go to
the "i" of "in" and type "cwout". Now you have changed the line to:

```
:e config.h.out ~
```

Now press <Enter> and this command will be executed. The command line window
will close.
   The <Enter> command will execute the line under the cursor. It doesn't
matter whether Vim is in Insert mode or in Normal mode.
   Changes in the command line window are lost. They do not result in the
history to be changed. Except that the command you execute will be added to
the end of the history, like with all executed commands.

The command line window is very useful when you want to have overview of the
history, lookup a similar command, change it a bit and execute it. A search
command can be used to find something.
   In the previous example the "?config" search command could have been used
to find the previous command that contains "config". It's a bit strange,
because you are using a command line to search in the command line window.
While typing that search command you can't open another command line window,
there can be only one.

==

Next chapter: |usr_21.txt| Go away and come back

usr_21.txt   For Vim version 8.0. Last change: 2012 Nov 02

                     VIM USER MANUAL - by Bram Moolenaar

                           Go away and come back

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

     Next chapter: |usr_22.txt| Finding the file to edit
 Previous chapter: |usr_20.txt| Typing command-line commands quickly
Table of contents: |usr_toc.txt|

==
21.1 Suspend and resume

Like most Unix programs Vim can be suspended by pressing CTRL-Z. This stops
Vim and takes you back to the shell it was started in. You can then do any
other commands until you are bored with them. Then bring back Vim with the
"fg" command. >

```
CTRL-Z
{any sequence of shell commands}
fg
```

You are right back where you left Vim, nothing has changed.
   In case pressing CTRL-Z doesn't work, you can also use ":suspend".
Don't forget to bring Vim back to the foreground, you would lose any changes
that you made!

Only Unix has support for this. On other systems Vim will start a shell for
you. This also has the functionality of being able to execute shell commands.
But it's a new shell, not the one that you started Vim from.
   When you are running the GUI you can't go back to the shell where Vim was
started. CTRL-Z will minimize the Vim window instead.

==
21.2 Executing shell commands

To execute a single shell command from Vim use ":!{command}". For example, to
see a directory listing: >

        :!ls
        :!dir

The first one is for Unix, the second one for MS-Windows.
   Vim will execute the program. When it ends you will get a prompt to hit
<Enter>. This allows you to have a look at the output from the command before
returning to the text you were editing.
   The "!" is also used in other places where a program is run. Let's take
a look at an overview:

        :!{program}   execute {program}
        :r !{program}   execute {program} and read its output
        :w !{program}   execute {program} and send text to its input
        :[range]!{program}   filter text through {program}

Notice that the presence of a range before "!{program}" makes a big
difference. Without it executes the program normally, with the range a number
of text lines is filtered through the program.

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

        :shell

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

When using the GUI the shell will be using the Vim window for its input and
output. Since Vim is not a terminal emulator, this will not work perfectly.
If you have trouble, try toggling the 'guipty' option. If this still doesn't
work well enough, start a new terminal to run the shell in. For example with:
>
        :!xterm&

==
21.3 Remembering information; viminfo

After editing for a while you will have text in registers, marks in various
files, a command line history filled with carefully crafted commands. When
you exit Vim all of this is lost. But you can get it back!

The viminfo file is designed to store status information:

        Command-line and Search pattern history

```
        Text in registers
        Marks for various files
        The buffer list
        Global variables

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

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

        :set viminfo=string

The string specifies what to save.  The syntax of this string is an option
character followed by an argument.  The option/argument pairs are separated by
commas.
   Take a look at how you can build up your own viminfo string.  First, the '
option is used to specify how many files for which you save marks (a-z).  Pick
a nice even number for this option (1000, for instance).  Your command now
looks like this: >

        :set viminfo='1000

The f option controls whether global marks (A-Z and 0-9) are stored.  If this
option is 0, none are stored.  If it is 1 or you do not specify an f option,
the marks are stored.  You want this feature, so now you have this: >

        :set viminfo='1000,f1

The < option controls how many lines are saved for each of the registers.  By
default, all the lines are saved.  If 0, nothing is saved.  To avoid adding
thousands of lines to your viminfo file (which might never get used and makes
starting Vim slower) you use a maximum of 500 lines: >

        :set viminfo='1000,f1,<500
<
Other options you might want to use:
        :       number of lines to save from the command line history
        @       number of lines to save from the input line history
        /       number of lines to save from the search history
        r       removable media, for which no marks will be stored (can be
                used several times)
        !       global variables that start with an uppercase letter and
                don't contain lowercase letters
        h       disable 'hlsearch' highlighting when starting
        %       the buffer list (only restored when starting Vim without file
                arguments)
        c       convert the text using 'encoding'
        n       name used for the viminfo file (must be the last option)

See the 'viminfo' option and |viminfo-file| for more information.

When you run Vim multiple times, the last one exiting will store its
information.  This may cause information that previously exiting Vims stored
to be lost.  Each item can be remembered only once.


GETTING BACK TO WHERE YOU STOPPED VIM

You are halfway editing a file and it's time to leave for holidays.  You exit
```

```
Vim and go enjoy yourselves, forgetting all about your work.  After a couple
of weeks you start Vim, and type:
>
        '0

And you are right back where you left Vim.  So you can get on with your work.
   Vim creates a mark each time you exit Vim.  The last one is '0.  The
position that '0 pointed to is made '1.  And '1 is made to '2, and so forth.
Mark '9 is lost.
   The |:marks| command is useful to find out where '0 to '9 will take you.


GETTING BACK TO SOME FILE

If you want to go back to a file that you edited recently, but not when
exiting Vim, there is a slightly more complicated way.  You can see a list of
files by typing the command: >

        :oldfiles
<       1: ~/.viminfo ~
        2: ~/text/resume.txt ~
        3: /tmp/draft ~

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

        :e #<2

Instead of ":e" you can use any command that has a file name argument, the
"#<2" item works in the same place as "%" (current file name) and "#"
(alternate file name).  So you can also split the window to edit the third
file: >

        :split #<3

That #<123 thing is a bit complicated when you just want to edit a file.
Fortunately there is a simpler way: >

        :browse oldfiles
<       1: ~/.viminfo ~
        2: ~/text/resume.txt ~
        3: /tmp/draft ~
        -- More --

You get the same list of files as with |:oldfiles|.  If you want to edit
"resume.txt" first press "q" to stop the listing.  You will get a prompt:

        Type number and <Enter> (empty cancels): ~

Type "2" and press <Enter> to edit the second file.

More info at |:oldfiles|, |v:oldfiles| and |c_#<|.


MOVE INFO FROM ONE VIM TO ANOTHER

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

        :wviminfo! ~/tmp/viminfo
```

```
And in the second Vim do: >

        :rviminfo! ~/tmp/viminfo

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

==============================================================================
*21.4*  Sessions

Suppose you are editing along, and it is the end of the day.  You want to quit
work and pick up where you left off the next day.  You can do this by saving
your editing session and restoring it the next day.
   A Vim session contains all the information about what you are editing.
This includes things such as the file list, window layout, global variables,
options and other information.  (Exactly what is remembered is controlled by
the 'sessionoptions' option, described below.)
   The following command creates a session file: >

        :mksession vimbook.vim

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

        :source vimbook.vim

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

        vim -S vimbook.vim

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

The windows that were open are restored, with the same position and size as
before.  Mappings and option values are like before.
   What exactly is restored depends on the 'sessionoptions' option.  The
default value is "blank,buffers,curdir,folds,help,options,winsize".

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

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

        :set sessionoptions+=resize


SESSION HERE, SESSION THERE
```

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

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

This first uses ":wall" to write all modified files. Then the current session
is saved, using ":mksession!". This overwrites the previous session. The
next time you load the secret session you can continue where you were at this
point. And finally you load the new "boring" session.

If you open help windows, split and close various windows, and generally mess
up the window layout, you can go back to the last saved session: >

```
:source ~/.vim/boring.vim
```

Thus you have complete control over whether you want to continue next time
where you are now, by saving the current setup in a session, or keep the
session file as a starting point.
   Another way of using sessions is to create a window layout that you like to
use, and save this in a session. Then you can go back to this layout whenever
you want.
   For example, this is a nice layout to use:

```
+----------------------------------------+
|                  VIM - main help file  |
|                                        |
|Move around:  Use the cursor keys, or "h|
|help.txt================================|
|explorer   |                            |
|dir        |~                           |
|dir        |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|~/=========|[No File]===================|
|                                        |
+----------------------------------------+
```

This has a help window at the top, so that you can read this text. The narrow
vertical window on the left contains a file explorer. This is a Vim plugin
that lists the contents of a directory. You can select files to edit there.
More about this in the next chapter.
   Create this from a just started Vim with: >

```
:help
CTRL-W w
:vertical split ~/
```

You can resize the windows a bit to your liking. Then save the session with:
>

```
:mksession ~/.vim/mine.vim
```

Now you can start Vim with this layout: >

```
vim -S ~/.vim/mine.vim
```

Hint: To open a file you see listed in the explorer window in the empty
window, move the cursor to the filename and press "O". Double clicking with
the mouse will also do this.

UNIX AND MS-WINDOWS

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

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

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

SESSIONS AND VIMINFO

Sessions store many things, but not the position of marks, contents of
registers and the command line history. You need to use the viminfo feature
for these things.
   In most situations you will want to use sessions separately from viminfo.
This can be used to switch to another session, but keep the command line
history. And yank text into registers in one session, and paste it back in
another session.
   You might prefer to keep the info with the session. You will have to do
this yourself then. Example: >

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

And to restore this again: >

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

==
21.5 Views

A session stores the looks of the whole of Vim. When you want to store the
properties for one window only, use a view.
   The use of a view is for when you want to edit a file in a specific way.
For example, you have line numbers enabled with the 'number' option and
defined a few folds. Just like with sessions, you can remember this view on
the file and restore it later. Actually, when you store a session, it stores
the view of each window.
   There are two basic ways to use views. The first is to let Vim pick a name
for the view file. You can restore the view when you later edit the same
file. To store the view for the current window: >

```
:mkview
```

Vim will decide where to store the view. When you later edit the same file
you get the view back with this command: >

```
:loadview
```

That's easy, isn't it?

Now you want to view the file without the 'number' option on, or with all folds open, you can set the options to make the window look that way. Then store this view with: >

```
:mkview 1
```

Obviously, you can get this back with: >

```
:loadview 1
```

Now you can switch between the two views on the file by using ":loadview" with and without the "1" argument.
You can store up to ten views for the same file this way, one unnumbered and nine numbered 1 to 9.

A VIEW WITH A NAME

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

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

You can restore it with: >

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

==
21.6 Modelines

When editing a specific file, you might set options specifically for that file. Typing these commands each time is boring. Using a session or view for editing a file doesn't work when sharing the file between several people.
The solution for this situation is adding a modeline to the file. This is a line of text that tells Vim the values of options, to be used in this file only.
A typical example is a C program where you make indents by a multiple of 4 spaces. This requires setting the 'shiftwidth' option to 4. This modeline will do that:

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

Put this line as one of the first or last five lines in the file. When editing the file, you will notice that 'shiftwidth' will have been set to four. When editing another file, it's set back to the default value of eight.
For some files the modeline fits well in the header, thus it can be put at the top of the file. For text files and other files where the modeline gets in the way of the normal contents, put it at the end of the file.

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

```
:set modelines=10
```

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

```
        :set nomodeline

Use this format for the modeline:

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

The "any-text" indicates that you can put any text before and after the part
that Vim will use.  This allows making it look like a comment, like what was
done above with /* and */.
   The " vim:" part is what makes Vim recognize this line.  There must be
white space before "vim", or "vim" must be at the start of the line.  Thus
using something like "gvim:" will not work.
   The part between the colons is a ":set" command.  It works the same way as
typing the ":set" command, except that you need to insert a backslash before a
colon (otherwise it would be seen as the end of the modeline).

Another example:

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

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

For more details see |modeline|.

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

Next chapter: |usr_22.txt|  Finding the file to edit

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_22.txt*    For Vim version 8.0.  Last change: 2016 Dec 13

                      VIM USER MANUAL - by Bram Moolenaar

                            Finding the file to edit


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

|22.1|  The file browser
|22.2|  The current directory
|22.3|  Finding a file
|22.4|  The buffer list

     Next chapter: |usr_23.txt|  Editing other files
 Previous chapter: |usr_21.txt|  Go away and come back
Table of contents: |usr_toc.txt|

==============================================================================
*22.1*  The file browser

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

        :edit .

Through the magic of autocommands and Vim scripts, the window will be filled
with the contents of the directory.  It looks like this:
```

```
" ============================================================================== ~
" Netrw Directory Listing                                        (netrw v109) ~
"   Sorted by      name ~
"   Sort sequence: [\/]$,\.h$,\.c$,\.cpp$,*,\.info$,\.swp$,\.o$\.obj$,\.bak$ ~
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:exec ~
" ============================================================================== ~
../ ~
./ ~
check/ ~
Makefile ~
autocmd.txt ~
change.txt ~
eval.txt~ ~
filetype.txt~ ~
help.txt.info ~
```

You can see these items:

1. The name of the browsing tool and its version number
2. The name of the browsing directory
3. The method of sorting (may be by name, time, or size)
4. How names are to be sorted (directories first, then *.h files, *.c files, etc)
5. How to get help (use the <F1> key), and an abbreviated listing of available commands
6. A listing of files, including "../", which allows one to list the parent directory.

If you have syntax highlighting enabled, the different parts are highlighted so as to make it easier to spot them.

You can use Normal mode Vim commands to move around in the text. For example, move the cursor atop a file and press <Enter>; you will then be editing that file. To go back to the browser use ":edit ." again, or use ":Explore". CTRL-O also works.

Try using <Enter> while the cursor is atop a directory name. The result is that the file browser moves into that directory and displays the items found there. Pressing <Enter> on the first directory "../" moves you one level higher. Pressing "-" does the same thing, without the need to move to the "../" item first.

You can press <F1> to get help on the things you can do in the netrw file browser. This is what you get: >

```
    9. Directory Browsing        netrw-browse   netrw-dir   netrw-list   netrw-help

    MAPS                                                                 netrw-maps
         <F1>.............Help.....................................|netrw-help|
         <cr>.............Browsing.................................|netrw-cr|
         <del>............Deleting Files or Directories.............|netrw-delete|
         -................Going Up.................................|netrw--|
         a................Hiding Files or Directories...............|netrw-a|
         mb...............Bookmarking a Directory...................|netrw-mb|
         gb...............Changing to a Bookmarked Directory.........|netrw-gb|
         c................Make Browsing Directory The Current Dir....|netrw-c|
         d................Make A New Directory......................|netrw-d|
         D................Deleting Files or Directories.............|netrw-D|
         <c-h>............Edit File/Directory Hiding List............|netrw-ctrl-h|
         i................Change Listing Style......................|netrw-i|
         <c-l>............Refreshing the Listing....................|netrw-ctrl-l|
         o................Browsing with a Horizontal Split..........|netrw-o|
```

```
         p................Use Preview Window.......................|netrw-p|
         P................Edit in Previous Window...................|netrw-p|
         q................Listing Bookmarks and History.............|netrw-qb|
         r................Reversing Sorting Order...................|netrw-r|
<       (etc)

The <F1> key thus brings you to a netrw directory browsing contents help page.
It's a regular help page; use the usual |CTRL-]| to jump to tagged help items
and |CTRL-O| to jump back.

To select files for display and editing: (with the cursor is atop a filename)

        <enter>         Open the file in the current window.       |netrw-cr|
        o               Horizontally split window and display file |netrw-o|
        v               Vertically split window and display file   |netrw-v|
        p               Use the |preview-window|                   |netrw-p|
        P               Edit in the previous window                |netrw-P|
        t               Open file in a new tab                     |netrw-t|

The following normal-mode commands may be used to control the browser display:

        i               Controls listing style (thin, long, wide, and tree).
                        The long listing includes size and date information.
        s               Repeatedly pressing s will change the way the files
                        are sorted; one may sort on name, modification time,
                        or size.
        r               Reverse the sorting order.

As a sampling of extra normal-mode commands:

        c               Change Vim's notion of the current directory to be
                        the same as the browser directory.  (see
                        |g:netrw_keepdir| to control this, too)
        R               Rename the file or directory under the cursor; a
                        prompt will be issued for the new name.
        D               Delete the file or directory under the cursor; a
                        confirmation request will be issued.
        mb gb           Make bookmark/goto bookmark


One may also use command mode; again, just a sampling:

        :Explore [directory]    Browse specified/current directory
        :NetrwSettings          A comprehensive list of your current netrw
                                settings with help linkage.

The netrw browser is not limited to just your local machine; one may use
urls such as:    (that trailing / is important)

        :Explore ftp://somehost/path/to/dir/
        :e scp://somehost/path/to/dir/

See |netrw-browse| for more.

==============================================================================
*22.2*  The current directory

Just like the shell, Vim has the concept of a current directory.  Suppose you
are in your home directory and want to edit several files in a directory
"VeryLongFileName".  You could do: >

        :edit VeryLongFileName/file1.txt
```

```
        :edit VeryLongFileName/file2.txt
        :edit VeryLongFileName/file3.txt

To avoid much of the typing, do this: >

        :cd VeryLongFileName
        :edit file1.txt
        :edit file2.txt
        :edit file3.txt

The ":cd" command changes the current directory.  You can see what the current
directory is with the ":pwd" command: >

        :pwd
        /home/Bram/VeryLongFileName

Vim remembers the last directory that you used.  Use "cd -" to go back to it.
Example: >

        :pwd
        /home/Bram/VeryLongFileName
        :cd /etc
        :pwd
        /etc
        :cd -
        :pwd
        /home/Bram/VeryLongFileName
        :cd -
        :pwd
        /etc


WINDOW LOCAL DIRECTORY

When you split a window, both windows use the same current directory.  When
you want to edit a number of files somewhere else in the new window, you can
make it use a different directory, without changing the current directory in
the other window.  This is called a local directory. >

        :pwd
        /home/Bram/VeryLongFileName
        :split
        :lcd /etc
        :pwd
        /etc
        CTRL-W w
        :pwd
        /home/Bram/VeryLongFileName

So long as no ":lcd" command has been used, all windows share the same current
directory.  Doing a ":cd" command in one window will also change the current
directory of the other window.
   For a window where ":lcd" has been used a different current directory is
remembered.  Using ":cd" or ":lcd" in other windows will not change it.
   When using a ":cd" command in a window that uses a different current
directory, it will go back to using the shared directory.

==============================================================================
*22.3*  Finding a file

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

```
        #include "inits.h" ~
```

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

```
        gf
```

Vim will find the file and edit it.
   What if the file is not in the current directory? Vim will use the 'path' option to find the file. This option is a list of directory names where to look for your file.
   Suppose you have your include files located in "c:/prog/include". This command will add it to the 'path' option: >

```
        :set path+=c:/prog/include
```

This directory is an absolute path. No matter where you are, it will be the same place. What if you have located files in a subdirectory, below where the file is? Then you can specify a relative path name. This starts with a dot: >

```
        :set path+=./proto
```

This tells Vim to look in the directory "proto", below the directory where the file in which you use "gf" is. Thus using "gf" on "inits.h" will make Vim look for "proto/inits.h", starting in the directory of the file.
   Without the "./", thus "proto", Vim would look in the "proto" directory below the current directory. And the current directory might not be where the file that you are editing is located.

The 'path' option allows specifying the directories where to search for files in many more ways. See the help on the 'path' option.
   The 'isfname' option is used to decide which characters are included in the file name, and which ones are not (e.g., the " character in the example above).

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

```
        :find inits.h
```

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

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

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

```
        vim "+find stdio.h"
```

This finds the file "stdio.h" in your value of 'path'. The quotes are necessary to have one argument |-+c|.

==
22.4 The buffer list

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

HIDDEN BUFFERS

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

```
        :hide edit two.txt
```

The buffer "one.txt" disappears from the screen, but Vim still knows that you
are editing this buffer, so it keeps the modified text. This is called a
hidden buffer: The buffer contains text, but you can't see it.
   The argument of ":hide" is another command. ":hide" makes that command
behave as if the 'hidden' option was set. You could also set this option
yourself. The effect is that when any buffer is abandoned, it becomes hidden.
   Be careful! When you have hidden buffers with changes, don't exit Vim
without making sure you have saved all the buffers.

INACTIVE BUFFERS

   When a buffer has been used once, Vim remembers some information about it.
When it is not displayed in a window and it is not hidden, it is still in the
buffer list. This is called an inactive buffer. Overview:

```
   Active               Appears in a window, text loaded.
   Hidden               Not in a window, text loaded.
   Inactive             Not in a window, no text loaded.
```

The inactive buffers are remembered, because Vim keeps information about them,
like marks. And remembering the file name is useful too, so that you can see
which files you have edited. And edit them again.

LISTING BUFFERS

View the buffer list with this command: >

```
        :buffers
```

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

```
        :ls
```

The output could look like this:

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

The first column contains the buffer number. You can use this to edit the
buffer without having to type the name, see below.
   After the buffer number come the flags. Then the name of the file
and the line number where the cursor was the last time.
   The flags that can appear are these (from left to right):

```
        u       Buffer is unlisted |unlisted-buffer|.
         %      Current buffer.
         #      Alternate buffer.
```

```
          a     Buffer is loaded and displayed.
          h     Buffer is loaded but hidden.
           =    Buffer is read-only.
           -    Buffer is not modifiable, the 'modifiable' option is off.
            +   Buffer has been modified.


EDITING A BUFFER

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

        :buffer 2

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

        :buffer help

Vim will find the best match for the name you type.  If there is only one
buffer that matches the name, it will be used.  In this case "help.txt".
   To open a buffer in a new window: >

        :sbuffer 3

This works with a name as well.


USING THE BUFFER LIST

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

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

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

        :bdelete 3

Again, this also works with a name.
   If you delete a buffer that was active (visible in a window), that window
will be closed.  If you delete the current buffer, the current window will be
closed.  If it was the last window, Vim will find another buffer to edit.  You
can't be editing nothing!

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

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

Next chapter: |usr_23.txt|  Editing other files

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_23.txt*    For Vim version 8.0.  Last change: 2006 Apr 24
```

Editing other files

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

Next chapter: |usr_24.txt| Inserting quickly
Previous chapter: |usr_22.txt| Finding the file to edit
Table of contents: |usr_toc.txt|

==========

23.1 DOS, Mac and Unix files

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

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

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

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

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

You will notice the format in the message you get when editing a file. You don't see anything if you edit a native file format. Thus editing a Unix file on Unix won't result in a remark. But when you edit a dos file, Vim will notify you of this:

```
"/tmp/test" [dos] 3L, 71C ~
```

For a Mac file you would see "[mac]".

The detected file format is stored in the 'fileformat' option. To see which format you have, execute the following command: >

```
:set fileformat?
```

The three names that Vim uses are:

```
unix            <LF>
dos             <CR><LF>
mac             <CR>
```

USING THE MAC FORMAT

On Unix, <LF> is used to break a line. It's not unusual to have a <CR>

character halfway a line. Incidentally, this happens quite often in Vi (and
Vim) scripts.
   On the Macintosh, where <CR> is the line break character, it's possible to
have a <LF> character halfway a line.
   The result is that it's not possible to be 100% sure whether a file
containing both <CR> and <LF> characters is a Mac or a Unix file. Therefore,
Vim assumes that on Unix you probably won't edit a Mac file, and doesn't check
for this type of file. To check for this format anyway, add "mac" to
'fileformats': >

        :set fileformats+=mac

Then Vim will take a guess at the file format. Watch out for situations where
Vim guesses wrong.

OVERRULING THE FORMAT

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

        :edit ++ff=unix file.txt

The "++" string is an item that tells Vim that an option name follows, which
overrules the default for this single command. "++ff" is used for
'fileformat'. You could also use "++ff=mac" or "++ff=dos".
   This doesn't work for any option, only "++ff" and "++enc" are currently
implemented. The full names "++fileformat" and "++encoding" also work.

CONVERSION

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

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

        :set fileformat=unix
        :write

The file is written in Unix format.

==
23.2 Files on the internet

Someone sends you an e-mail message, which refers to a file by its URL. For
example:

        You can find the information here: ~
                ftp://ftp.vim.org/pub/vim/README ~

You could start a program to download the file, save it on your local disk and
then start Vim to edit it.
   There is a much simpler way. Move the cursor to any character of the URL.
Then use this command: >

```
        gf

With a bit of luck, Vim will figure out which program to use for downloading
the file, download it and edit the copy.  To open the file in a new window use
CTRL-W f.
   If something goes wrong you will get an error message.  It's possible that
the URL is wrong, you don't have permission to read it, the network connection
is down, etc.  Unfortunately, it's hard to tell the cause of the error.  You
might want to try the manual way of downloading the file.

Accessing files over the internet works with the netrw plugin.  Currently URLs
with these formats are recognized:

        ftp://          uses ftp
        rcp://          uses rcp
        scp://          uses scp
        http://         uses wget (reading only)

Vim doesn't do the communication itself, it relies on the mentioned programs
to be available on your computer.  On most Unix systems "ftp" and "rcp" will
be present.  "scp" and "wget" might need to be installed.

Vim detects these URLs for each command that starts editing a new file, also
with ":edit" and ":split", for example.  Write commands also work, except for
http://.

For more information, also about passwords, see |netrw|.

==============================================================================
*23.3*  Encryption

Some information you prefer to keep to yourself.  For example, when writing
a test on a computer that students also use.  You don't want clever students
to figure out a way to read the questions before the exam starts.  Vim can
encrypt the file for you, which gives you some protection.
   To start editing a new file with encryption, use the "-x" argument to start
Vim.  Example: >

        vim -x exam.txt

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

        Enter encryption key: ~

Carefully type the secret key now.  You cannot see the characters you type,
they will be replaced by stars.  To avoid the situation that a typing mistake
will cause trouble, Vim asks you to enter the key again:

        Enter same key again: ~

You can now edit this file normally and put in all your secrets.  When you
finish editing the file and tell Vim to exit, the file is encrypted and
written.
   When you edit the file with Vim, it will ask you to enter the same key
again.  You don't need to use the "-x" argument.  You can also use the normal
":edit" command.  Vim adds a magic string to the file by which it recognizes
that the file was encrypted.
   If you try to view this file using another program, all you get is garbage.
Also, if you edit the file with Vim and enter the wrong key, you get garbage.
Vim does not have a mechanism to check if the key is the right one (this makes
it much harder to break the key).
```

SWITCHING ENCRYPTION ON AND OFF

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

The next time you write the file this will be done without encryption.
Setting the 'key' option to enable encryption is not a good idea, because the password appears in the clear. Anyone shoulder-surfing can read your password.
To avoid this problem, the ":X" command was created. It asks you for an encryption key, just like the "-x" argument did: >

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

LIMITS ON ENCRYPTION

The encryption algorithm used by Vim is weak. It is good enough to keep out the casual prowler, but not good enough to keep out a cryptology expert with lots of time on his hands. Also you should be aware that the swap file is not encrypted; so while you are editing, people with superuser privileges can read the unencrypted text from this file.
One way to avoid letting people read your swap file is to avoid using one. If the -n argument is supplied on the command line, no swap file is used (instead, Vim puts everything in memory). For example, to edit the encrypted file "file.txt" without a swap file use the following command: >

```
vim -x -n file.txt
```

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

```
:setlocal noswapfile
```

Since there is no swapfile, recovery will be impossible. Save the file a bit more often to avoid the risk of losing your changes.

While the file is in memory, it is in plain text. Anyone with privilege can look in the editor's memory and discover the contents of the file.
If you use a viminfo file, be aware that the contents of text registers are written out in the clear as well.
If you really want to secure the contents of a file, edit it only on a portable computer not connected to a network, use good encryption tools, and keep the computer locked up in a big safe when not in use.

==
23.4 Binary files

You can edit binary files with Vim. Vim wasn't really made for this, thus there are a few restrictions. But you can read a file, change a character and write it back, with the result that only that one character was changed and the file is identical otherwise.
To make sure that Vim does not use its clever tricks in the wrong way, add the "-b" argument when starting Vim: >

```
vim -b datafile
```

This sets the 'binary' option. The effect of this is that unexpected side effects are turned off. For example, 'textwidth' is set to zero, to avoid

automatic formatting of lines. And files are always read in Unix file format.

Binary mode can be used to change a message in a program. Be careful not to insert or delete any characters, it would stop the program from working. Use "R" to enter replace mode.

Many characters in the file will be unprintable. To see them in Hex format: >

```
:set display=uhex
```

Otherwise, the "ga" command can be used to see the value of the character under the cursor. The output, when the cursor is on an <Esc>, looks like this:

```
<^[>  27,  Hex 1b,  Octal 033 ~
```

There might not be many line breaks in the file. To get some overview switch the 'wrap' option off: >

```
:set nowrap
```

BYTE POSITION

To see on which byte you are in the file use this command: >

```
g CTRL-G
```

The output is verbose:

```
    Col 9-16 of 9-16; Line 277 of 330; Word 1806 of 2058; Byte 10580 of 12206 ~
```

The last two numbers are the byte position in the file and the total number of bytes. This takes into account how 'fileformat' changes the number of bytes that a line break uses.
    To move to a specific byte in the file, use the "go" command. For example, to move to byte 2345: >

```
2345go
```

USING XXD

A real binary editor shows the text in two ways: as it is and in hex format. You can do this in Vim by first converting the file with the "xxd" program. This comes with Vim.
   First edit the file in binary mode: >

```
vim -b datafile
```

Now convert the file to a hex dump with xxd: >

```
:%!xxd
```

The text will look like this:

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

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

```
changed, or the other way around.
   Finally convert it back with:
>
        :%!xxd -r

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

See the manual page of xxd for more information.

==============================================================================
*23.5*  Compressed files

This is easy: You can edit a compressed file just like any other file.  The
"gzip" plugin takes care of decompressing the file when you edit it.  And
compressing it again when you write it.
   These compression methods are currently supported:

        .Z      compress
        .gz     gzip
        .bz2    bzip2

Vim uses the mentioned programs to do the actual compression and
decompression.  You might need to install the programs first.

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

Next chapter: |usr_24.txt|  Inserting quickly

Copyright: see |manual-copyright|  vim:tw=78:ts=8:ft=help:norl:
*usr_24.txt*    For Vim version 8.0.  Last change: 2006 Jul 23

                     VIM USER MANUAL - by Bram Moolenaar

                             Inserting quickly


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

|24.1|  Making corrections
|24.2|  Showing matches
|24.3|  Completion
|24.4|  Repeating an insert
|24.5|  Copying from another line
|24.6|  Inserting a register
|24.7|  Abbreviations
|24.8|  Entering special characters
|24.9|  Digraphs
|24.10| Normal mode commands

     Next chapter: |usr_25.txt|  Editing formatted text
 Previous chapter: |usr_23.txt|  Editing other files
Table of contents: |usr_toc.txt|

==============================================================================
*24.1*  Making corrections

The <BS> key was already mentioned.  It deletes the character just before the
cursor.  The <Del> key does the same for the character under (after) the
```

cursor.
   When you typed a whole word wrong, use CTRL-W:

```
        The horse had fallen to the sky ~
                                      CTRL-W
        The horse had fallen to the ~
```

If you really messed up a line and want to start over, use CTRL-U to delete it. This keeps the text after the cursor and the indent. Only the text from the first non-blank to the cursor is deleted. With the cursor on the "f" of "fallen" in the next line pressing CTRL-U does this:

```
        The horse had fallen to the ~
                      CTRL-U
        fallen to the ~
```

When you spot a mistake a few words back, you need to move the cursor there to correct it. For example, you typed this:

```
        The horse had follen to the ground ~
```

You need to change "follen" to "fallen". With the cursor at the end, you would type this to correct it: >

```
                                        <Esc>4blraA

<       get out of Insert mode          <Esc>
        four words back                      4b
        move on top of the "o"                 l
        replace with "a"                        ra
        restart Insert mode                       A
```

Another way to do this: >

```
                <C-Left><C-Left><C-Left><C-Left><Right><Del>a<End>

<       four words back             <C-Left><C-Left><C-Left><C-Left>
        move on top of the "o"                  <Right>
        delete the "o"                                 <Del>
        insert an "a"                                       a
        go to end of the line                                <End>
```

This uses special keys to move around, while remaining in Insert mode. This resembles what you would do in a modeless editor. It's easier to remember, but takes more time (you have to move your hand from the letters to the cursor keys, and the <End> key is hard to press without looking at the keyboard).
   These special keys are most useful when writing a mapping that doesn't leave Insert mode. The extra typing doesn't matter then.
   An overview of the keys you can use in Insert mode:

```
        <C-Home>        to start of the file
        <PageUp>        a whole screenful up
        <Home>          to start of line
        <S-Left>        one word left
        <C-Left>        one word left
        <S-Right>       one word right
        <C-Right>       one word right
        <End>           to end of the line
        <PageDown>      a whole screenful down
        <C-End>         to end of the file
```

There are a few more, see |ins-special-special|.

==
24.2 Showing matches

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

        :set showmatch

When you now type a text like "(example)", as soon as you type the) Vim will
briefly move the cursor to the matching (, keep it there for half a second,
and move back to where you were typing.
   In case there is no matching (, Vim will beep. Then you know that you
might have forgotten the (somewhere, or typed a) too many.
   The match will also be shown for [] and {} pairs. You don't have to wait
with typing the next character, as soon as Vim sees it the cursor will move
back and inserting continues as before.
   You can change the time Vim waits with the 'matchtime' option. For
example, to make Vim wait one and a half second: >

        :set matchtime=15

The time is specified in tenths of a second.

==
24.3 Completion

Vim can automatically complete words on insertion. You type the first part of
a word, press CTRL-P, and Vim guesses the rest.
   Suppose, for example, that you are creating a C program and want to type in
the following:

        total = ch_array[0] + ch_array[1] + ch_array[2]; ~

You start by entering the following:

        total = ch_array[0] + ch_ ~

At this point, you tell Vim to complete the word using the command CTRL-P.
Vim searches for a word that starts with what's in front of the cursor. In
this case, it is "ch_", which matches with the word ch_array. So typing
CTRL-P gives you the following:

        total = ch_array[0] + ch_array ~

After a little more typing, you get this (ending in a space):

        total = ch_array[0] + ch_array[1] + ~

If you now type CTRL-P Vim will search again for a word that completes the
word before the cursor. Since there is nothing in front of the cursor, it
finds the first word backwards, which is "ch_array". Typing CTRL-P again
gives you the next word that matches, in this case "total". A third CTRL-P
searches further back. If there is nothing else, it causes the editor to run
out of words, so it returns to the original text, which is nothing. A fourth
CTRL-P causes the editor to start over again with "ch_array".

To search forward, use CTRL-N. Since the search wraps around the end of the
file, CTRL-N and CTRL-P will find the same matches, but in a different
sequence. Hint: CTRL-N is Next-match and CTRL-P is Previous-match.

The Vim editor goes through a lot of effort to find words to complete. By

default, it searches the following places:

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

OPTIONS

You can customize the search order with the 'complete' option.

The 'ignorecase' option is used. When it is set, case differences are ignored when searching for matches.

A special option for completion is 'infercase'. This is useful to find matches while ignoring case ('ignorecase' must be set) but still using the case of the word typed so far. Thus if you type "For" and Vim finds a match "fortunately", it will result in "Fortunately".

COMPLETING SPECIFIC ITEMS

If you know what you are looking for, you can use these commands to complete with a certain type of item:

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

After each of them CTRL-N can be used to find the next match, CTRL-P to find the previous match.
More information for each of these commands here: |ins-completion|.

COMPLETING FILE NAMES

Let's take CTRL-X CTRL-F as an example. This will find file names. It scans the current directory for files and displays each one that matches the word in front of the cursor.
Suppose, for example, that you have the following files in the current directory:

```
main.c  sub_count.c  sub_done.c  sub_exit.c
```

Now enter Insert mode and start typing:

```
The exit code is in the file sub ~
```

At this point, you enter the command CTRL-X CTRL-F. Vim now completes the current word "sub" by looking at the files in the current directory. The first match is sub_count.c. This is not the one you want, so you match the next file by typing CTRL-N. This match is sub_done.c. Typing CTRL-N again takes you to sub_exit.c. The results:

```
        The exit code is in the file sub_exit.c ~

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

        the file is found in /usr/ ~

If you now press CTRL-N you go back to "/u".  Instead, to accept the "/usr/"
and go one directory level deeper, use CTRL-X CTRL-F again:

        the file is found in /usr/X11R6/ ~

The results depend on what is found in your file system, of course.  The
matches are sorted alphabetically.


COMPLETING IN SOURCE CODE

Source code files are well structured.  That makes it possible to do
completion in an intelligent way.  In Vim this is called Omni completion.  In
some other editors it's called intellisense, but that is a trademark.

The key to Omni completion is CTRL-X CTRL-O.  Obviously the O stands for Omni
here, so that you can remember it easier.  Let's use an example for editing C
source:

        { ~
            struct foo *p; ~
            p-> ~

The cursor is after "p->".  Now type CTRL-X CTRL-O.  Vim will offer you a list
of alternatives, which are the items that "struct foo" contains.  That is
quite different from using CTRL-P, which would complete any word, while only
members of "struct foo" are valid here.

For Omni completion to work you may need to do some setup.  At least make sure
filetype plugins are enabled.  Your vimrc file should contain a line like
this: >
        filetype plugin on
Or: >
        filetype plugin indent on

For C code you need to create a tags file and set the 'tags' option.  That is
explained |ft-c-omni|.  For other filetypes you may need to do something
similar, look below |compl-omni-filetypes|.  It only works for specific
filetypes.  Check the value of the 'omnifunc' option to find out if it would
work.

==============================================================================
*24.4*  Repeating an insert

If you press CTRL-A, the editor inserts the text you typed the last time you
were in Insert mode.
   Assume, for example, that you have a file that begins with the following:

        "file.h" ~
        /* Main program begins */ ~

You edit this file by inserting "#include " at the beginning of the first
line:
```

```
        #include "file.h" ~
        /* Main program begins */ ~
```

You go down to the beginning of the next line using the commands "j^". You now start to insert a new "#include" line. So you type: >

```
        i CTRL-A
```

The result is as follows:

```
        #include "file.h" ~
        #include /* Main program begins */ ~
```

The "#include " was inserted because CTRL-A inserts the text of the previous insert. Now you type "main.h"<Enter> to finish the line:

```
        #include "file.h" ~
        #include "main.h" ~
        /* Main program begins */ ~
```

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

==

24.5 Copying from another line

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

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

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

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

Now you type "prev":

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

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

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

Now type "prev;" to finish it off.

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

==

24.6 Inserting a register

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

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

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

```
        "vyiw
```

"v is the register specification, "yiw" is yank-inner-word. Now edit the file where the new line is to be inserted, and type the first letters:

```
        r = ~
```

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

```
        r = VeryLongFunction ~
```

You continue to type the characters in between the function name, and use CTRL-R v two times more.
   You could have done the same with completion. Using a register is useful when there are many words that start with the same characters.

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

==
24.7 Abbreviations

An abbreviation is a short word that takes the place of a long one. For example, "ad" stands for "advertisement". Vim enables you to type an abbreviation and then will automatically expand it for you.
   To tell Vim to expand "ad" into "advertisement" every time you insert it, use the following command: >

```
        :iabbrev ad advertisement
```

Now, when you type "ad", the whole word "advertisement" will be inserted into the text. This is triggered by typing a character that can't be part of a word, for example a space:

```
        What Is Entered         What You See
        I saw the a             I saw the a ~
        I saw the ad            I saw the ad ~
        I saw the ad<Space>     I saw the advertisement<Space> ~
```

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

ABBREVIATING SEVERAL WORDS

It is possible to define an abbreviation that results in multiple words. For example, to define "JB" as "Jack Benny", use the following command: >

```
        :iabbrev JB Jack Benny
```

As a programmer, I use two rather unusual abbreviations: >

```
        :iabbrev #b /**************************************
        :iabbrev #e <Space>**************************************/
```

These are used for creating boxed comments. The comment starts with #b, which draws the top line. I then type the comment text and use #e to draw the bottom line.
   Notice that the #e abbreviation begins with a space. In other words, the first two characters are space-star. Usually Vim ignores spaces between the abbreviation and the expansion. To avoid that problem, I spell space as seven characters: <, S, p, a, c, e, >.

```
        Note:
        ":iabbrev" is a long word to type.  ":iab" works just as well.
        That's abbreviating the abbreviate command!
```

FIXING TYPING MISTAKES

It's very common to make the same typing mistake every time. For example, typing "teh" instead of "the". You can fix this with an abbreviation: >

```
        :abbreviate teh the
```

You can add a whole list of these. Add one each time you discover a common mistake.

LISTING ABBREVIATIONS

The ":abbreviate" command lists the abbreviations:

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

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

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

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

```
        :edit ad
```

DELETING ABBREVIATIONS

To get rid of an abbreviation, use the ":unabbreviate" command. Suppose you have the following abbreviation: >

```
        :abbreviate @f fresh
```

You can remove it with this command: >

```
        :unabbreviate @f
```

While you type this, you will notice that @f is expanded to "fresh". Don't worry about this, Vim understands it anyway (except when you have an

```
abbreviation for "fresh", but that's very unlikely).
   To remove all the abbreviations: >

        :abclear

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


REMAPPING ABBREVIATIONS

There is one thing to watch out for when defining an abbreviation: The
resulting string should not be mapped.  For example: >

        :abbreviate @a adder
        :imap dd disk-door

When you now type @a, you will get "adisk-doorer".  That's not what you want.
To avoid this, use the ":noreabbrev" command.  It does the same as
":abbreviate", but avoids that the resulting string is used for mappings: >

        :noreabbrev @a adder

Fortunately, it's unlikely that the result of an abbreviation is mapped.

==============================================================================
*24.8*  Entering special characters

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

        CTRL-V <Esc>

Inserts an escape character.  Thus you don't leave Insert mode.  (Don't type
the space after CTRL-V, it's only to make this easier to read).

        Note:
        On MS-Windows CTRL-V is used to paste text.  Use CTRL-Q instead of
        CTRL-V.  On Unix, on the other hand, CTRL-Q does not work on some
        terminals, because it has a special meaning.

You can also use the command CTRL-V {digits} to insert a character with the
decimal number {digits}.  For example, the character number 127 is the <Del>
character (but not necessarily the <Del> key!).  To insert <Del> type: >

        CTRL-V 127

You can enter characters up to 255 this way.  When you type fewer than two
digits, a non-digit will terminate the command.  To avoid the need of typing a
non-digit, prepend one or two zeros to make three digits.
   All the next commands insert a <Tab> and then a dot:

        CTRL-V 9.
        CTRL-V 09.
        CTRL-V 009.

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

        CTRL-V x7f
```

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

```
        CTRL-V o123
        CTRL-V u1234
        CTRL-V U12345678
```

==

24.9 Digraphs

Some characters are not on the keyboard. For example, the copyright character (\A9). To type these characters in Vim, you use digraphs, where two characters represent one. To enter a \A9, for example, you press three keys: >

```
        CTRL-K Co
```

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

```
        :digraphs
```

Vim will display the digraph table. Here are three lines of it:

```
  AC ~_ 159  NS |  160  !I \A1  161  Ct \A2  162  Pd \A3  163  Cu \A4  164  Ye \A5
165 ~
  BB \A6  166  SE \A7  167  ': \A8  168  Co \A9  169  -a \AA  170  << \AB  171  NO
\AC  172 ~
  -- \AD  173  Rg \AE  174  'm \AF  175  DG \B0  176  +- \B1  177  2S \B2  178  3S
\B3  179 ~
```

This shows, for example, that the digraph you get by typing CTRL-K Pd is the character (\A3). This is character number 163 (decimal).
   Pd is short for Pound. Most digraphs are selected to give you a hint about the character they will produce. If you look through the list you will understand the logic.
   You can exchange the first and second character, if there is no digraph for that combination. Thus CTRL-K dP also works. Since there is no digraph for "dP" Vim will also search for a "Pd" digraph.

```
        Note:
        The digraphs depend on the character set that Vim assumes you are
        using. On MS-DOS they are different from MS-Windows. Always use
        ":digraphs" to find out which digraphs are currently available.
```

You can define your own digraphs. Example: >

```
        :digraph a" \E4
```

This defines that CTRL-K a" inserts an \E4 character. You can also specify the character with a decimal number. This defines the same digraph: >

```
        :digraph a" 228
```

More information about digraphs here: |digraphs|
   Another way to insert special characters is with a keymap. More about that here: |45.5|

==

24.10 Normal mode commands

Insert mode offers a limited number of commands. In Normal mode you have many more. When you want to use one, you usually leave Insert mode with <Esc>,

execute the Normal mode command, and re-enter Insert mode with "i" or "a".
   There is a quicker way. With CTRL-O {command} you can execute any Normal
mode command from Insert mode. For example, to delete from the cursor to the
end of the line: >

        CTRL-O D

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

        CTRL-O "g3dw

This deletes up to the third word into register g.

==

Next chapter: |usr_25.txt| Editing formatted text

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_25.txt   For Vim version 8.0. Last change: 2016 Mar 28

                     VIM USER MANUAL - by Bram Moolenaar

                             Editing formatted text

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

|25.1| Breaking lines
|25.2| Aligning text
|25.3| Indents and tabs
|25.4| Dealing with long lines
|25.5| Editing tables

     Next chapter: |usr_26.txt| Repeating
 Previous chapter: |usr_24.txt| Inserting quickly
Table of contents: |usr_toc.txt|

==
25.1 Breaking lines

Vim has a number of functions that make dealing with text easier. By default,
the editor does not perform automatic line breaks. In other words, you have
to press <Enter> yourself. This is useful when you are writing programs where
you want to decide where the line ends. It is not so good when you are
creating documentation and want the text to be at most 70 character wide.
   If you set the 'textwidth' option, Vim automatically inserts line breaks.
Suppose, for example, that you want a very narrow column of only 30
characters. You need to execute the following command: >

        :set textwidth=30

Now you start typing (ruler added):

                  1   2   3
        123456789012345678901234567890123456
        I taught programming for a whi ~

If you type "l" next, this makes the line longer than the 30-character limit.
When Vim sees this, it inserts a line break and you get the following:

```
                1         2         3
        12345678901234567890123456789012345
        I taught programming for a ~
        whil ~
```

Continuing on, you can type in the rest of the paragraph:

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

You do not have to type newlines; Vim puts them in automatically.

```
        Note:
        The 'wrap' option makes Vim display lines with a line break, but this
        doesn't insert a line break in the file.
```

REFORMATTING

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

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

This does not look good. To get the paragraph into shape you use the "gq" operator.
   Let's first use this with a Visual selection. Starting from the first line, type: >

```
        v4jgq
```

"v" to start Visual mode, "4j" to move to the end of the paragraph and then the "gq" operator. The result is:

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

Note: there is a way to do automatic formatting for specific types of text layouts, see |auto-format|.

Since "gq" is an operator, you can use one of the three ways to select the text it works on: With Visual mode, with a movement and with a text object.
   The example above could also be done with "gq4j". That's less typing, but

```
you have to know the line count.  A more useful motion command is "}".  This
moves to the end of a paragraph.  Thus "gq}" formats from the cursor to the
end of the current paragraph.
   A very useful text object to use with "gq" is the paragraph.  Try this: >

        gqap

"ap" stands for "a-paragraph".  This formats the text of one paragraph
(separated by empty lines).  Also the part before the cursor.
   If you have your paragraphs separated by empty lines, you can format the
whole file by typing this: >

        gggqG

"gg" to move to the first line, "gqG" to format until the last line.
   Warning: If your paragraphs are not properly separated, they will be joined
together.  A common mistake is to have a line with a space or tab.  That's a
blank line, but not an empty line.

Vim is able to format more than just plain text.  See |fo-table| for how to
change this.  See the 'joinspaces' option to change the number of spaces used
after a full stop.
   It is possible to use an external program for formatting.  This is useful
if your text can't be properly formatted with Vim's builtin command.  See the
'formatprg' option.

==============================================================================
*25.2*  Aligning text

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

        :{range}center [width]

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

        :1,5center 40

results in the following:

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


RIGHT ALIGNMENT

Similarly, the ":right" command right-justifies the text: >

        :1,5right 37

gives this result:

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

LEFT ALIGNMENT

Finally there is this command: >

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

Unlike ":center" and ":right", however, the argument to ":left" is not the length of the line. Instead it is the left margin. If it is omitted, the text will be put against the left side of the screen (using a zero margin would do the same). If it is 5, the text will be indented 5 spaces. For example, use these commands: >

```
:1left 5
:2,5left
```

This results in the following:

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

JUSTIFYING TEXT

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

```
:packadd justify
```

Or put this line in your |vimrc|: >

```
packadd! justify
```

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

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

```
:%!fmt
```

==
25.3 Indents and tabs

Indents can be used to make text stand out from the rest. The example texts in this manual, for example, are indented by eight spaces or a tab. You would normally enter this by typing a tab at the start of each line. Take this text:

```
the first line ~
the second line ~
```

This is entered by typing a tab, some text, <Enter>, tab and more text.
The 'autoindent' option inserts indents automatically: >

```
:set autoindent
```

When a new line is started it gets the same indent as the previous line. In

the above example, the tab after the <Enter> is not needed anymore.

INCREASING INDENT

To increase the amount of indent in a line, use the ">" operator. Often this is used as ">>", which adds indent to the current line.
   The amount of indent added is specified with the 'shiftwidth' option. The default value is 8. To make ">>" insert four spaces worth of indent, for example, type this: >

```
:set shiftwidth=4
```

When used on the second line of the example text, this is what you get:

```
the first line ~
    the second line ~
```

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

TABSTOP

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

```
:set softtabstop=4
```

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

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

CHANGING TABS

You edit a file which was written with a tabstop of 3. In Vim it looks ugly, because it uses the normal tabstop value of 8. You can fix this by setting 'tabstop' to 3. But you have to do this every time you edit this file.
   Vim can change the use of tabstops in your file. First, set 'tabstop' to make the indents look good, then use the ":retab" command: >

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

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

==

25.4 Dealing with long lines

Sometimes you will be editing a file that is wider than the number of columns in the window. When that occurs, Vim wraps the lines so that everything fits on the screen.
   If you switch the 'wrap' option off, each line in the file shows up as one line on the screen. Then the ends of the long lines disappear off the screen to the right.
   When you move the cursor to a character that can't be seen, Vim will scroll the text to show it. This is like moving a viewport over the text in the horizontal direction.
   By default, Vim does not display a horizontal scrollbar in the GUI. If you want to enable one, use the following command: >

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

One horizontal scrollbar will appear at the bottom of the Vim window.

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

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

Let's attempt to show this with one line of text. The cursor is on the "w" of "which". The "current window" above the line indicates the text that is currently visible. The "window"s below the text indicate the text that is visible after the command left of it.

```
                            |<-- current window -->|
                some long text, part of which is visible in the window ~
ze                |<--     window     -->|
zH                 |<--     window     -->|
4zh                      |<--     window     -->|
zh                          |<--     window     -->|
zl                            |<--     window     -->|
4zl                              |<--     window     -->|
zL                                     |<--     window     -->|
zs                                    |<--     window     -->|
```

MOVING WITH WRAP OFF

When 'wrap' is off and the text has scrolled horizontally, you can use the following commands to move the cursor to a character you can see. Thus text left and right of the window is ignored. These never cause the text to scroll:

```
g0              to first visible character in this line
g^              to first non-blank visible character in this line
gm              to middle of this line
g$              to last visible character in this line

        |<--     window    -->|
some long    text, part of which is visible ~
```

```
                g0  g^    gm        g$
```

BREAKING AT WORDS *edit-no-break*

When preparing text for use by another program, you might have to make paragraphs without a line break. A disadvantage of using 'nowrap' is that you can't see the whole sentence you are working on. When 'wrap' is on, words are broken halfway, which makes them hard to read.

A good solution for editing this kind of paragraph is setting the 'linebreak' option. Vim then breaks lines at an appropriate place when displaying the line. The text in the file remains unchanged.

Without 'linebreak' text might look like this:

```
        +---------------------------------+
        |letter generation program for a b|
        |ank.  They wanted to send out a s|
        |pecial, personalized letter to th|
        |eir richest 1000 customers.  Unfo|
        |rtunately for the programmer, he |
        +---------------------------------+
```

After: >

```
        :set linebreak
```

it looks like this:

```
        +---------------------------------+
        |letter generation program for a  |
        |bank.  They wanted to send out a |
        |special, personalized letter to  |
        |their richest 1000 customers.    |
        |Unfortunately for the programmer,|
        +---------------------------------+
```

Related options:
'breakat' specifies the characters where a break can be inserted.
'showbreak' specifies a string to show at the start of broken line.
Set 'textwidth' to zero to avoid a paragraph to be split.

MOVING BY VISIBLE LINES

The "j" and "k" commands move to the next and previous lines. When used on a long line, this means moving a lot of screen lines at once.

To move only one screen line, use the "gj" and "gk" commands. When a line doesn't wrap they do the same as "j" and "k". When the line does wrap, they move to a character displayed one line below or above.

You might like to use these mappings, which bind these movement commands to the cursor keys: >

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

TURNING A PARAGRAPH INTO ONE LINE *edit-paragraph-join*

If you want to import text into a program like MS-Word, each paragraph should be a single line. If your paragraphs are currently separated with empty lines, this is how you turn each paragraph into a single line: >

```
        :g/./,/^$/join
```

That looks complicated. Let's break it up in pieces:

```
        :g/./           A ":global" command that finds all lines that contain
                        at least one character.
             ,/^$/      A range, starting from the current line (the non-empty
                        line) until an empty line.
                  join  The ":join" command joins the range of lines together
                        into one line.
```

Starting with this text, containing eight lines broken at column 30:

```
        +----------------------------------+
        |A letter generation program       |
        |for a bank.  They wanted to       |
        |send out a special,               |
        |personalized letter.              |
        |                                  |
        |To their richest 1000             |
        |customers.  Unfortunately for     |
        |the programmer,                   |
        +----------------------------------+
```

You end up with two lines:

```
        +----------------------------------+
        |A letter generation program for a |
        |bank.  They wanted to send out a s|
        |pecial, personalized letter.      |
        |To their richest 1000 customers.  |
        |Unfortunately for the programmer, |
        +----------------------------------+
```

Note that this doesn't work when the separating line is blank but not empty;
when it contains spaces and/or tabs. This command does work with blank lines:
>

```
        :g/\S/,/^\s*$/join
```

This still requires a blank or empty line at the end of the file for the last
paragraph to be joined.

==
25.5 Editing tables

Suppose you are editing a table with four columns:

```
        nice table        test 1        test 2      test 3 ~
        input A           0.534 ~
        input B           0.913 ~
```

You need to enter numbers in the third column. You could move to the second
line, use "A", enter a lot of spaces and type the text.
   For this kind of editing there is a special option: >

```
        set virtualedit=all
```

Now you can move the cursor to positions where there isn't any text. This is
called "virtual space". Editing a table is a lot easier this way.
   Move the cursor by searching for the header of the last column: >

```
        /test 3
```

Now press "j" and you are right where you can enter the value for "input A".
Typing "0.693" results in:

```
        nice table        test 1     test 2      test 3 ~
        input A           0.534                  0.693 ~
        input B           0.913 ~
```

Vim has automatically filled the gap in front of the new text for you. Now,
to enter the next field in this column use "Bj". "B" moves back to the start
of a white space separated word. Then "j" moves to the place where the next
field can be entered.

```
        Note:
        You can move the cursor anywhere in the display, also beyond the end
        of a line. But Vim will not insert spaces there, until you insert a
        character in that position.
```

COPYING A COLUMN

You want to add a column, which should be a copy of the third column and
placed before the "test 1" column. Do this in seven steps:
1. Move the cursor to the left upper corner of this column, e.g., with
   "/test 3".
2. Press CTRL-V to start blockwise Visual mode.
3. Move the cursor down two lines with "2j". You are now in "virtual space":
   the "input B" line of the "test 3" column.
4. Move the cursor right, to include the whole column in the selection, plus
   the space that you want between the columns. "9l" should do it.
5. Yank the selected rectangle with "y".
6. Move the cursor to "test 1", where the new column must be placed.
7. Press "P".

The result should be:

```
        nice table        test 3    test 1     test 2      test 3 ~
        input A           0.693     0.534                  0.693 ~
        input B                     0.913 ~
```

Notice that the whole "test 1" column was shifted right, also the line where
the "test 3" column didn't have text.

Go back to non-virtual cursor movements with: >

```
        :set virtualedit=
```

VIRTUAL REPLACE MODE

The disadvantage of using 'virtualedit' is that it "feels" different. You
can't recognize tabs or spaces beyond the end of line when moving the cursor
around. Another method can be used: Virtual Replace mode.
   Suppose you have a line in a table that contains both tabs and other
characters. Use "rx" on the first tab:

```
        inp     0.693   0.534   0.693 ~

              |
           rx |
              V

        inpx0.693   0.534       0.693 ~
```

The layout is messed up. To avoid that, use the "gr" command:

```
        inp     0.693   0.534   0.693 ~

                |
          grx   |
                V

        inpx    0.693   0.534   0.693 ~
```

What happens is that the "gr" command makes sure the new character takes the right amount of screen space. Extra spaces or tabs are inserted to fill the gap. Thus what actually happens is that a tab is replaced by "x" and then blanks added to make the text after it keep its place. In this case a tab is inserted.
   When you need to replace more than one character, you use the "R" command to go to Replace mode (see |04.9|). This messes up the layout and replaces the wrong characters:

```
        inp     0       0.534   0.693 ~

                |
         R0.786 |
                V

        inp     0.78634 0.693 ~
```

The "gR" command uses Virtual Replace mode. This preserves the layout:

```
        inp     0       0.534   0.693 ~

                |
        gR0.786 |
                V

        inp     0.786   0.534   0.693 ~
```

==

Next chapter: |usr_26.txt| Repeating

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_26.txt   For Vim version 8.0. Last change: 2006 Apr 24

VIM USER MANUAL - by Bram Moolenaar

Repeating

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

Next chapter: |usr_27.txt| Search commands and patterns
Previous chapter: |usr_25.txt| Editing formatted text
Table of contents: |usr_toc.txt|

```
==============================================================================
*26.1*  Repeating with Visual mode

Visual mode is very handy for making a change in any sequence of lines.  You
can see the highlighted text, thus you can check if the correct lines are
changed.  But making the selection takes some typing.  The "gv" command
selects the same area again.  This allows you to do another operation on the
same text.
   Suppose you have some lines where you want to change "2001" to "2002" and
"2000" to "2001":

        The financial results for 2001 are better ~
        than for 2000.  The income increased by 50%, ~
        even though 2001 had more rain than 2000. ~
                        2000            2001 ~
        income          45,403          66,234 ~

First change "2001" to "2002".  Select the lines in Visual mode, and use: >

        :s/2001/2002/g

Now use "gv" to reselect the same text.  It doesn't matter where the cursor
is.  Then use ":s/2000/2001/g" to make the second change.
   Obviously, you can repeat these changes several times.

==============================================================================
*26.2*  Add and subtract

When repeating the change of one number into another, you often have a fixed
offset.  In the example above, one was added to each year.  Instead of typing
a substitute command for each year that appears, the CTRL-A command can be
used.
   Using the same text as above, search for a year: >

        /19[0-9][0-9]\|20[0-9][0-9]

Now press CTRL-A.  The year will be increased by one:

        The financial results for 2002 are better ~
        than for 2000.  The income increased by 50%, ~
        even though 2001 had more rain than 2000. ~
                        2000            2001 ~
        income          45,403          66,234 ~

Use "n" to find the next year, and press "." to repeat the CTRL-A ("." is a
bit quicker to type).  Repeat "n" and "." for all years that appear.
   Hint: set the 'hlsearch' option to see the matches you are going to change,
then you can look ahead and do it faster.

Adding more than one can be done by prepending the number to CTRL-A.  Suppose
you have this list:

        1.  item four ~
        2.  item five ~
        3.  item six ~

Move the cursor to "1." and type: >

        3 CTRL-A

The "1." will change to "4.".  Again, you can use "." to repeat this on the
```

other numbers.

Another example:

```
        006     foo bar ~
        007     foo bar ~
```

Using CTRL-A on these numbers results in:

```
        007     foo bar ~
        010     foo bar ~
```

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

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

The CTRL-X command does subtraction in a similar way.

==
26.3 Making a change in many files

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

```
        :args *.c
```
<
This finds all C files and edits the first one. Now you can perform a substitution command on all these files: >

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

The ":argdo" command takes an argument that is another command. That command will be executed on all files in the argument list.
   The "%s" substitute command that follows works on all lines. It finds the word "x_cnt" with "\<x_cnt\>". The "\<" and "\>" are used to match the whole word only, and not "px_cnt" or "x_cnt2".
   The flags for the substitute command include "g" to replace all occurrences of "x_cnt" in the same line. The "e" flag is used to avoid an error message when "x_cnt" does not appear in the file. Otherwise ":argdo" would abort on the first file where "x_cnt" was not found.
   The "|" separates two commands. The following "update" command writes the file only if it was changed. If no "x_cnt" was changed to "x_counter" nothing happens.

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

==
26.4 Using Vim from a shell script

Suppose you have a lot of files in which you need to change the string "-person-" to "Jones" and then print it. How do you do that? One way is to do a lot of typing. The other is to write a shell script to do the work.
   The Vim editor does a superb job as a screen-oriented editor when using Normal mode commands. For batch processing, however, Normal mode commands do

```
not result in clear, commented command files; so here you will use Ex mode
instead.  This mode gives you a nice command-line interface that makes it easy
to put into a batch file.  ("Ex command" is just another name for a
command-line (:) command.)
   The Ex mode commands you need are as follows: >

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

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

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

The for-done loop is a shell construct to repeat the two lines in between,
while the $file variable is set to a different file name each time.
   The second line runs the Vim editor in Ex mode (-e argument) on the file
$file and reads commands from the file "change.vim".  The -s argument tells
Vim to operate in silent mode.  In other words, do not keep outputting the
:prompt, or any other prompt for that matter.
   The "lpr -r tempfile" command prints the resulting "tempfile" and deletes
it (that's what the -r argument does).


READING FROM STDIN

Vim can read text on standard input.  Since the normal way is to read commands
there, you must tell Vim to read text instead.  This is done by passing the
"-" argument in place of a file.  Example: >

        ls | vim -

This allows you to edit the output of the "ls" command, without first saving
the text in a file.
   If you use the standard input to read text from, you can use the "-S"
argument to read a script: >

        producer | vim -S change.vim -


NORMAL MODE SCRIPTS

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

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

The commands in "script" are executed like you typed them.  Don't forget that
a line break is interpreted as pressing <Enter>.  In Normal mode that moves
the cursor to the next line.
   To create the script you can edit the script file and type the commands.
You need to imagine what the result would be, which can be a bit difficult.
```

Another way is to record the commands while you perform them manually. This
is how you do that: >

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

All typed keys will be written to "script". If you make a small mistake you
can just continue and remember to edit the script later.
The "-w" argument appends to an existing script. That is good when you
want to record the script bit by bit. If you want to start from scratch and
start all over, use the "-W" argument. It overwrites any existing file.

==

Next chapter: |usr_27.txt| Search commands and patterns

usr_27.txt For Vim version 8.0. Last change: 2010 Mar 28

VIM USER MANUAL - by Bram Moolenaar

Search commands and patterns

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

Next chapter: |usr_28.txt| Folding
Previous chapter: |usr_26.txt| Repeating
Table of contents: |usr_toc.txt|

==
27.1 Ignoring case

By default, Vim's searches are case sensitive. Therefore, "include",
"INCLUDE", and "Include" are three different words and a search will match
only one of them.
Now switch on the 'ignorecase' option: >

```
:set ignorecase
```

Search for "include" again, and now it will match "Include", "INCLUDE" and
"InClUDe". (Set the 'hlsearch' option to quickly see where a pattern
matches.)
You can switch this off again with: >

```
:set noignorecase
```

But let's keep it set, and search for "INCLUDE". It will match exactly the
same text as "include" did. Now set the 'smartcase' option: >

```
:set ignorecase smartcase
```

If you have a pattern with at least one uppercase character, the search
becomes case sensitive. The idea is that you didn't have to type that
uppercase character, so you must have done it because you wanted case to
match. That's smart!
    With these two options set you find the following matches:

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

CASE IN ONE PATTERN

If you want to ignore case for one specific pattern, you can do this by
prepending the "\c" string. Using "\C" will make the pattern to match case.
This overrules the 'ignorecase' and 'smartcase' options, when "\c" or "\C" is
used their value doesn't matter.

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

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

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

        Note:
        If your search takes much longer than you expected, you can interrupt
        it with CTRL-C on Unix and CTRL-Break on MS-DOS and MS-Windows.

==
27.2 Wrapping around the file end

By default, a forward search starts searching for the given string at the
current cursor location. It then proceeds to the end of the file. If it has
not found the string by that time, it starts from the beginning and searches
from the start of the file to the cursor location.
   Keep in mind that when repeating the "n" command to search for the next
match, you eventually get back to the first match. If you don't notice this
you keep searching forever! To give you a hint, Vim displays this message:

        search hit BOTTOM, continuing at TOP ~

If you use the "?" command, to search in the other direction, you get this
message:

        search hit TOP, continuing at BOTTOM ~

Still, you don't know when you are back at the first match. One way to see
this is by switching on the 'ruler' option: >

```
        :set ruler

Vim will display the cursor position in the lower righthand corner of the
window (in the status line if there is one).  It looks like this:

        101,29       84% ~

The first number is the line number of the cursor.  Remember the line number
where you started, so that you can check if you passed this position again.


NOT WRAPPING

To turn off search wrapping, use the following command: >

        :set nowrapscan

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

        E385: search hit BOTTOM without match for: forever ~

Thus you can find all matches by going to the start of the file with "gg" and
keep searching until you see this message.
   If you search in the other direction, using "?", you get:

        E384: search hit TOP without match for: forever ~

==============================================================================
*27.3*  Offsets

By default, the search command leaves the cursor positioned on the beginning
of the pattern.  You can tell Vim to leave it some other place by specifying
an offset.  For the forward search command "/", the offset is specified by
appending a slash (/) and the offset: >

        /default/2

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

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


CHARACTER OFFSETS

The "e" offset indicates an offset from the end of the match.  It moves the
cursor onto the last character of the match.  The command: >

        /const/e

puts the cursor on the "t" of "const".
   From that position, adding a number moves forward that many characters.
This command moves to the character just after the match: >

        /const/e+1

A positive number moves the cursor to the right, a negative number moves it to
```

the left. For example: >

```
	/const/e-1
```

moves the cursor to the "s" of "const".

If the offset begins with "b", the cursor moves to the beginning of the pattern. That's not very useful, since leaving out the "b" does the same thing. It does get useful when a number is added or subtracted. The cursor then goes forward or backward that many characters. For example: >

```
	/const/b+2
```

Moves the cursor to the beginning of the match and then two characters to the right. Thus it lands on the "n".

REPEATING

To repeat searching for the previously used search pattern, but with a different offset, leave out the pattern: >

```
	/that
	//e
```

Is equal to: >

```
	/that/e
```

To repeat with the same offset: >

```
	/
```

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

```
	//
```

SEARCHING BACKWARDS

The "?" command uses offsets in the same way, but you must use "?" to separate the offset from the pattern, instead of "/": >

```
	?const?e-2
```

The "b" and "e" keep their meaning, they don't change direction with the use of "?".

START POSITION

When starting a search, it normally starts at the cursor position. When you specify a line offset, this can cause trouble. For example: >

```
	/const/-2
```

This finds the next word "const" and then moves two lines up. If you use "n" to search again, Vim could start at the current position and find the same "const" match. Then using the offset again, you would be back where you started. You would be stuck!
   It could be worse: Suppose there is another match with "const" in the next line. Then repeating the forward search would find this match and move two

lines up. Thus you would actually move the cursor back!

When you specify a character offset, Vim will compensate for this. Thus the
search starts a few characters forward or backward, so that the same match
isn't found again.

==
27.4 Matching multiple times

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

	/a*

matches "a", "aa", "aaa", etc. But also "" (the empty string), because zero
times is included.
   The "*" only applies to the item directly before it. Thus "ab*" matches
"a", "ab", "abb", "abbb", etc. To match a whole string multiple times, it
must be grouped into one item. This is done by putting "\(" before it and
"\)" after it. Thus this command: >

	/\(ab\)*

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

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

	/ab\+

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

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

	/folders\=

Matches "folder" and "folders".

SPECIFIC COUNTS

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

	/ab\{3,5}

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

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

MATCHING AS LITTLE AS POSSIBLE

The items so far match as many characters as they can find. To match as few as possible, use "\{-n,m}". It works the same as "\{n,m}", except that the minimal amount possible is used.
   For example, use: >

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

Will match "ab" in "abbb". Actually, it will never match more than one b, because there is no reason to match more. It requires something else to force it to match more than the lower limit.
   The same rules apply to removing "n" and "m". It's even possible to remove both of the numbers, resulting in "\{-}". This matches the item before it zero or more times, as few as possible. The item by itself always matches zero times. It is useful when combined with something else. Example: >

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

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

```
/a.*b
```

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

==========
27.5 Alternatives

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

```
/foo\|bar
```

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

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

Matches "one", "two" and "three".
   To match multiple times, the whole thing must be placed in "\(" and "\)": >

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

This matches "foo", "foobar", "foofoo", "barfoobar", etc.
   Another example: >

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

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

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

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

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

==========
27.6 Character ranges

To match "a", "b" or "c" you could use "/a\|b\|c". When you want to match all letters from "a" to "z" this gets very long. There is a shorter method: >

```
/[a-z]
```

The [] construct matches a single character. Inside you specify which
characters to match. You can include a list of characters, like this: >

```
/[0123456789abcdef]
```

This will match any of the characters included. For consecutive characters
you can specify the range. "0-3" stands for "0123". "w-z" stands for "wxyz".
Thus the same command as above can be shortened to: >

```
/[0-9a-f]
```

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

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

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

COMPLEMENTED RANGE

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

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

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

PREDEFINED RANGES

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

```
/\a
```

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

```
item    matches                 equivalent ~
\d      digit                   [0-9]
\D      non-digit               [^0-9]
\x      hex digit               [0-9a-fA-F]
\X      non-hex digit           [^0-9a-fA-F]
\s      white space             [       ]     (<Tab> and <Space>)
\S      non-white characters    [^      ]     (not <Tab> and <Space>)
\l      lowercase alpha         [a-z]
\L      non-lowercase alpha     [^a-z]
\u      uppercase alpha         [A-Z]
\U      non-uppercase alpha     [^A-Z]
```

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

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

==============================================================================
*27.7*  Character classes

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

        /\f\+

The "\f" items stands for file name characters.  Thus this matches a sequence
of characters that can be a file name.
   Which characters can be part of a file name depends on the system you are
using.  On MS-Windows, the backslash is included, on Unix it is not.  This is
specified with the 'isfname' option.  The default value for Unix is: >

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

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

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

The character classes are:

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

==============================================================================
*27.8*  Matching a line break

Vim can find a pattern that includes a line break.  You need to specify where
the line break happens, because all items mentioned so far don't match a line
break.
   To check for a line break in a specific place, use the "\n" item: >

        /the\nword

This will match at a line that ends in "the" and the next line starts with
"word".  To match "the word" as well, you need to match a space or a line
```

break. The item to use for it is "_s": >

```
/the\_sword
```

To allow any amount of white space: >

```
/the\_s\+word
```

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

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

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

```
Note:
"\_.*" matches everything until the end of the file.  Be careful with
this, it can make a search command very slow.
```

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

```
/"\_[^"]*"
```

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

==

27.9 Examples

Here are a few search patterns you might find useful. This shows how the items mentioned above can be combined.

FINDING A CALIFORNIA LICENSE PLATE

A sample license plate number is "1MGU103". It has one digit, three uppercase letters and three digits. Directly putting this into a search pattern: >

```
/\d\u\u\u\d\d\d
```

Another way is to specify that there are three digits and letters with a count: >

```
/\d\u\{3}\d\{3}
```

Using [] ranges instead: >

```
/[0-9][A-Z]\{3}[0-9]\{3}
```

Which one of these you should use? Whichever one you can remember. The simple way you can remember is much faster than the fancy way that you can't. If you can remember them all, then avoid the last one, because it's both more typing and slower to execute.

FINDING AN IDENTIFIER

In C programs (and many other computer languages) an identifier starts with a letter and further consists of letters and digits. Underscores can be used

too. This can be found with: >

```
        /\<\h\w*\>
```

"\<" and "\>" are used to find only whole words. "\h" stands for "[A-Za-z_]"
and "\w" for "[0-9A-Za-z_]".

        Note:
        "\<" and "\>" depend on the 'iskeyword' option. If it includes "-",
        for example, then "ident-" is not matched. In this situation use: >

```
                /\w\@<!\h\w*\w\@!
```
<
        This checks if "\w" does not match before or after the identifier.
        See |/\@<!| and |/\@!|.

==

Next chapter: |usr_28.txt| Folding

usr_28.txt   For Vim version 8.0. Last change: 2008 Jun 14

                     VIM USER MANUAL - by Bram Moolenaar

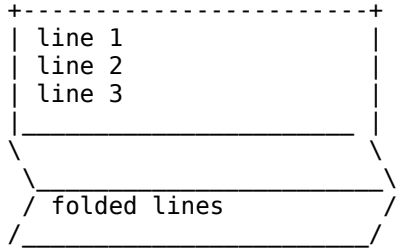
                                  Folding

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

     Next chapter: |usr_29.txt| Moving through programs
 Previous chapter: |usr_27.txt| Search commands and patterns
Table of contents: |usr_toc.txt|

==
28.1 What is folding?

Folding is used to show a range of lines in the buffer as a single line on the
screen. Like a piece of paper which is folded to make it shorter:



```
        | line 12               |
        | line 13               |
        | line 14               |
        +-----------------------+
```

The text is still in the buffer, unchanged. Only the way lines are displayed
is affected by folding.

The advantage of folding is that you can get a better overview of the
structure of text, by folding lines of a section and replacing it with a line
that indicates that there is a section.

==
28.2 Manual folding

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

```
        zfap
```

You will see that the paragraph is replaced by a highlighted line. You have
created a fold. |zf| is an operator and |ap| a text object selection. You
can use the |zf| operator with any movement command to create a fold for the
text that it moved over. |zf| also works in Visual mode.

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

```
        zo
```

And you can close the fold again with: >

```
        zc
```

All the folding commands start with "z". With some fantasy, this looks like a
folded piece of paper, seen from the side. The letter after the "z" has a
mnemonic meaning to make it easier to remember the commands:

```
        zf      F-old creation
        zo      O-pen a fold
        zc      C-lose a fold
```

Folds can be nested: A region of text that contains folds can be folded
again. For example, you can fold each paragraph in this section, and then
fold all the sections in this chapter. Try it out. You will notice that
opening the fold for the whole chapter will restore the nested folds as they
were, some may be open and some may be closed.

Suppose you have created several folds, and now want to view all the text.
You could go to each fold and type "zo". To do this faster, use this command: >

```
        zr
```

This will R-educe the folding. The opposite is: >

```
        zm
```

This folds M-ore. You can repeat "zr" and "zm" to open and close nested folds
of several levels.

If you have nested several levels deep, you can open all of them with: >

```
        zR
```

This R-educes folds until there are none left. And you can close all folds
with: >

	zM

This folds M-ore and M-ore.

You can quickly disable the folding with the |zn| command. Then |zN| brings
back the folding as it was. |zi| toggles between the two. This is a useful
way of working:
- create folds to get overview on your file
- move around to where you want to do your work
- do |zi| to look at the text and edit it
- do |zi| again to go back to moving around

More about manual folding in the reference manual: |fold-manual|

==
28.3 Working with folds

When some folds are closed, movement commands like "j" and "k" move over a
fold like it was a single, empty line. This allows you to quickly move around
over folded text.

You can yank, delete and put folds as if it was a single line. This is very
useful if you want to reorder functions in a program. First make sure that
each fold contains a whole function (or a bit less) by selecting the right
'foldmethod'. Then delete the function with "dd", move the cursor and put it
with "p". If some lines of the function are above or below the fold, you can
use Visual selection:
- put the cursor on the first line to be moved
- hit "V" to start Visual mode
- put the cursor on the last line to be moved
- hit "d" to delete the selected lines.
- move the cursor to the new position and "p"ut the lines there.

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

	:set foldcolumn=4

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

You can use the mouse to open a fold by clicking on the "+" in the foldcolumn.
Clicking on the "-" or a "|" below it will close an open fold.

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

When in Insert mode, the fold at the cursor line is never closed. That allows
you to see what you type!

Folds are opened automatically when jumping around or moving the cursor left
or right. For example, the "0" command opens the fold under the cursor
(if 'foldopen' contains "hor", which is the default). The 'foldopen' option
can be changed to open folds for specific commands. If you want the line
under the cursor always to be open, do this: >

```
        :set foldopen=all

Warning: You won't be able to move onto a closed fold then.  You might want to
use this only temporarily and then set it back to the default: >

        :set foldopen&

You can make folds close automatically when you move out of it: >

        :set foldclose=all

This will re-apply 'foldlevel' to all folds that don't contain the cursor.
You have to try it out if you like how this feels.  Use |zm| to fold more and
|zr| to fold less (reduce folds).

The folding is local to the window.  This allows you to open two windows on
the same buffer, one with folds and one without folds.  Or one with all folds
closed and one with all folds open.

==============================================================================
*28.4*  Saving and restoring folds

When you abandon a file (starting to edit another one), the state of the folds
is lost.  If you come back to the same file later, all manually opened and
closed folds are back to their default.  When folds have been created
manually, all folds are gone!  To save the folds use the |:mkview| command: >

        :mkview

This will store the settings and other things that influence the view on the
file.  You can change what is stored with the 'viewoptions' option.
When you come back to the same file later, you can load the view again: >

        :loadview

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

        :mkview 3
        :loadview 2

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

==============================================================================
*28.5*  Folding by indent

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

Try this by setting the 'foldmethod' option: >

        :set foldmethod=indent

Then you can use the |zm| and |zr| commands to fold more and reduce folding.
It's easy to see on this example text:

This line is not indented
```

```
        This line is indented once
                This line is indented twice
                This line is indented twice
        This line is indented once
This line is not indented
        This line is indented once
        This line is indented once
```

Note that the relation between the amount of indent and the fold depth depends on the 'shiftwidth' option. Each 'shiftwidth' worth of indent adds one to the depth of the fold. This is called a fold level.

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

```
        :set foldlevel=3
```

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

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

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

This can be combined: You can first close most folds by using |zm| a few times and then open a specific fold with |zo|. Or open all folds with |zR| and then close specific folds with |zc|.

But you cannot manually define folds when 'foldmethod' is "indent", as that would conflict with the relation between the indent and the fold level.

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

==
28.6 Folding with markers

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

Try it: >

```
        :set foldmethod=marker
```

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

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

Notice that the folded line will display the text before the marker. This is very useful to tell what the fold contains.

It's quite annoying when the markers don't pair up correctly after moving some lines around. This can be avoided by using numbered markers. Example:

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

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

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

At every numbered marker a fold at the specified level begins. This will make any fold at a higher level stop here. You can just use numbered start markers to define all folds. Only when you want to explicitly stop a fold before another starts you need to add an end marker.

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

==
28.7 Folding by syntax

For each language Vim uses a different syntax file. This defines the colors for various items in the file. If you are reading this in Vim, in a terminal that supports colors, the colors you see are made with the "help" syntax file.
In the syntax files it is possible to add syntax items that have the "fold" argument. These define a fold region. This requires writing a syntax file and adding these items in it. That's not so easy to do. But once it's done, all folding happens automatically.
Here we'll assume you are using an existing syntax file. Then there is nothing more to explain. You can open and close folds as explained above. The folds will be created and deleted automatically when you edit the file.

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

==
28.8 Folding by expression

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

```
        :set foldmethod=expr
        :set foldexpr=strlen(substitute(substitute(getline(v:lnum),'\\s','',
\"g\"),'[^>].*','',''))
```

You can try it out on this text:

```
> quoted text he wrote
> quoted text he wrote
> > double quoted text I wrote
> > double quoted text I wrote
```

Explanation for the 'foldexpr' used in the example (inside out):

```
   getline(v:lnum)                      gets the current line
   substitute(...,'\\s','','g')         removes all white space from the line
   substitute(...,'[^>].*','','')       removes everything after leading '>'s
   strlen(...)                          counts the length of the string, which
                                        is the number of '>'s found
```

Note that a backslash must be inserted before every space, double quote and backslash for the ":set" command. If this confuses you, do >

```
        :set foldexpr
```

to check the actual resulting value. To correct a complicated expression, use the command-line completion: >

```
        :set foldexpr=<Tab>
```

Where <Tab> is a real Tab. Vim will fill in the previous value, which you can then edit.

When the expression gets more complicated you should put it in a function and set 'foldexpr' to call that function.

More about folding by expression in the reference manual: |fold-expr|

==
28.9 Folding unchanged lines

This is useful when you set the 'diff' option in the same window. The |vimdiff| command does this for you. Example: >

```
        :setlocal diff foldmethod=diff scrollbind nowrap foldlevel=1
```

Do this in every window that shows a different version of the same file. You will clearly see the differences between the files, while the text that didn't change is folded.

For more details see |fold-diff|.

==
28.10 Which fold method to use?

All these possibilities make you wonder which method you should choose. Unfortunately, there is no golden rule. Here are some hints.

If there is a syntax file with folding for the language you are editing, that is probably the best choice. If there isn't one, you might try to write it. This requires a good knowledge of search patterns. It's not easy, but when it's working you will not have to define folds manually.

Typing commands to manually fold regions can be used for unstructured text. Then use the |:mkview| command to save and restore your folds.

The marker method requires you to change the file. If you are sharing the files with other people or you have to meet company standards, you might not be allowed to add them.
   The main advantage of markers is that you can put them exactly where you want them. That avoids that a few lines are missed when you cut and paste folds. And you can add a comment about what is contained in the fold.

Folding by indent is something that works in many files, but not always very well. Use it when you can't use one of the other methods. However, it is very useful for outlining. Then you specifically use one 'shiftwidth' for

each nesting level.

Folding with expressions can make folds in almost any structured text. It is quite simple to specify, especially if the start and end of a fold can easily be recognized.
   If you use the "expr" method to define folds, but they are not exactly how you want them, you could switch to the "manual" method. This will not remove the defined folds. Then you can delete or add folds manually.

==

Next chapter: |usr_29.txt| Moving through programs

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_29.txt For Vim version 8.0. Last change: 2016 Feb 27

VIM USER MANUAL - by Bram Moolenaar

Moving through programs

The creator of Vim is a computer programmer. It's no surprise that Vim contains many features to aid in writing programs. Jump around to find where identifiers are defined and used. Preview declarations in a separate window. There is more in the next chapter.

Next chapter: |usr_30.txt| Editing programs
Previous chapter: |usr_28.txt| Folding
Table of contents: |usr_toc.txt|

==
29.1 Using tags

What is a tag? It is a location where an identifier is defined. An example is a function definition in a C or C++ program. A list of tags is kept in a tags file. This can be used by Vim to directly jump from any place to the tag, the place where an identifier is defined.
   To generate the tags file for all C files in the current directory, use the following command: >

```
ctags *.c
```

"ctags" is a separate program. Most Unix systems already have it installed. If you do not have it yet, you can find Exuberant ctags here:

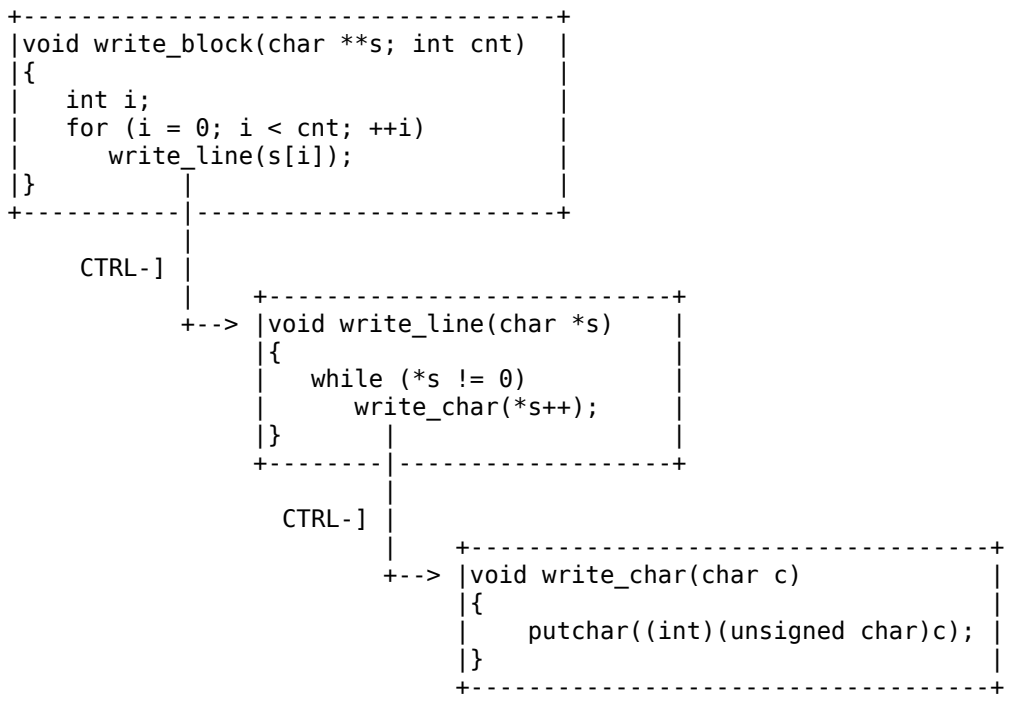
```
http://ctags.sf.net ~
```

Now when you are in Vim and you want to go to a function definition, you can jump to it by using the following command: >

```
:tag startlist
```

This command will find the function "startlist" even if it is in another file.
   The CTRL-] command jumps to the tag of the word that is under the cursor. This makes it easy to explore a tangle of C code. Suppose, for example, that you are in the function "write_block". You can see that it calls

"write_line". But what does "write_line" do? By placing the cursor on the call to "write_line" and pressing CTRL-], you jump to the definition of this function.
   The "write_line" function calls "write_char". You need to figure out what it does. So you position the cursor over the call to "write_char" and press CTRL-]. Now you are at the definition of "write_char".



The ":tags" command shows the list of tags that you traversed through:

```
:tags
  # TO tag         FROM line  in file/text ~
  1  1 write_line          8  write_block.c ~
  2  1 write_char          7  write_line.c ~
> ~
>
```

Now to go back. The CTRL-T command goes to the preceding tag. In the example above you get back to the "write_line" function, in the call to "write_char".
   This command takes a count argument that indicates how many tags to jump back. You have gone forward, and now back. Let's go forward again. The following command goes to the tag on top of the list: >

```
:tag
```

You can prefix it with a count and jump forward that many tags. For example: ":3tag". CTRL-T also can be preceded with a count.
   These commands thus allow you to go down a call tree with CTRL-] and back up again with CTRL-T. Use ":tags" to find out where you are.

SPLIT WINDOWS

The ":tag" command replaces the file in the current window with the one containing the new function. But suppose you want to see not only the old function but also the new one? You can split the window using the ":split" command followed by the ":tag" command. Vim has a shorthand command that does both: >

```
        :stag tagname

To split the current window and jump to the tag under the cursor use this
command: >

        CTRL-W ]

If a count is specified, the new window will be that many lines high.


MORE TAGS FILES

When you have files in many directories, you can create a tags file in each of
them.  Vim will then only be able to jump to tags within that directory.
   To find more tags files, set the 'tags' option to include all the relevant
tags files.  Example: >

        :set tags=./tags,./../tags,./*/tags

This finds a tags file in the same directory as the current file, one
directory level higher and in all subdirectories.
   This is quite a number of tags files, but it may still not be enough.  For
example, when editing a file in "~/proj/src", you will not find the tags file
"~/proj/sub/tags".  For this situation Vim offers to search a whole directory
tree for tags files.  Example: >

        :set tags=~/proj/**/tags


ONE TAGS FILE

When Vim has to search many places for tags files, you can hear the disk
rattling.  It may get a bit slow.  In that case it's better to spend this
time while generating one big tags file.  You might do this overnight.
   This requires the Exuberant ctags program, mentioned above.  It offers an
argument to search a whole directory tree: >

        cd ~/proj
        ctags -R .

The nice thing about this is that Exuberant ctags recognizes various file
types.  Thus this doesn't work just for C and C++ programs, also for Eiffel
and even Vim scripts.  See the ctags documentation to tune this.
   Now you only need to tell Vim where your big tags file is: >

        :set tags=~/proj/tags


MULTIPLE MATCHES

When a function is defined multiple times (or a method in several classes),
the ":tag" command will jump to the first one.  If there is a match in the
current file, that one is used first.
   You can now jump to other matches for the same tag with: >

        :tnext

Repeat this to find further matches.  If there are many, you can select which
one to jump to: >

        :tselect tagname
```

Vim will present you with a list of choices:

```
  # pri kind tag               file ~
  1 F   f    mch_init          os_amiga.c ~
               mch_init() ~
  2 F   f    mch_init          os_mac.c ~
               mch_init() ~
  3 F   f    mch_init          os_msdos.c ~
               mch_init(void) ~
  4 F   f    mch_init          os_riscos.c ~
               mch_init() ~
Enter nr of choice (<CR> to abort):  ~
```

You can now enter the number (in the first column) of the match that you would like to jump to. The information in the other columns give you a good idea of where the match is defined.

To move between the matching tags, these commands can be used:

```
:tfirst                 go to first match
:[count]tprevious       go to [count] previous match
:[count]tnext           go to [count] next match
:tlast                  go to last match
```

If [count] is omitted then one is used.

GUESSING TAG NAMES

Command line completion is a good way to avoid typing a long tag name. Just type the first bit and press <Tab>: >

```
:tag write_<Tab>
```

You will get the first match. If it's not the one you want, press <Tab> until you find the right one.

Sometimes you only know part of the name of a function. Or you have many tags that start with the same string, but end differently. Then you can tell Vim to use a pattern to find the tag.

Suppose you want to jump to a tag that contains "block". First type this: >

```
:tag /block
```

Now use command line completion: press <Tab>. Vim will find all tags that contain "block" and use the first match.

The "/" before a tag name tells Vim that what follows is not a literal tag name, but a pattern. You can use all the items for search patterns here. For example, suppose you want to select a tag that starts with "write_": >

```
:tselect /^write_
```

The "^" specifies that the tag starts with "write_". Otherwise it would also be found halfway a tag name. Similarly "$" at the end makes sure the pattern matches until the end of a tag.

A TAGS BROWSER

Since CTRL-] takes you to the definition of the identifier under the cursor, you can use a list of identifier names as a table of contents. Here is an example.

First create a list of identifiers (this requires Exuberant ctags): >

```
ctags --c-types=f -f functions *.c
```

Now start Vim without a file, and edit this file in Vim, in a vertically split window: >

```
vim
:vsplit functions
```

The window contains a list of all the functions. There is some more stuff, but you can ignore that. Do ":setlocal ts=99" to clean it up a bit.
In this window, define a mapping: >

```
:nnoremap <buffer> <CR> 0ye<C-W>w:tag <C-R>"<CR>
```

Move the cursor to the line that contains the function you want to go to. Now press <Enter>. Vim will go to the other window and jump to the selected function.

RELATED ITEMS

To make case in tag names be ignored, you can set 'ignorecase' while leaving 'tagcase' as "followic", or set 'tagcase' to "ignore".

The 'tagbsearch' option tells if the tags file is sorted or not. The default is to assume a sorted tags file, which makes a tags search a lot faster, but doesn't work if the tags file isn't sorted.

The 'taglength' option can be used to tell Vim the number of significant characters in a tag.

Cscope is a free program. It does not only find places where an identifier is declared, but also where it is used. See |cscope|.

==========

29.2 The preview window

When you edit code that contains a function call, you need to use the correct arguments. To know what values to pass you can look at how the function is defined. The tags mechanism works very well for this. Preferably the definition is displayed in another window. For this the preview window can be used.
To open a preview window to display the function "write_char": >

```
:ptag write_char
```

Vim will open a window, and jumps to the tag "write_char". Then it takes you back to the original position. Thus you can continue typing without the need to use a CTRL-W command.
If the name of a function appears in the text, you can get its definition in the preview window with: >

```
CTRL-W }
```

There is a script that automatically displays the text where the word under the cursor was defined. See |CursorHold-example|.

To close the preview window use this command: >

```
:pclose
```

To edit a specific file in the preview window, use ":pedit". This can be
useful to edit a header file, for example: >

```
:pedit defs.h
```

Finally, ":psearch" can be used to find a word in the current file and any
included files and display the match in the preview window. This is
especially useful when using library functions, for which you do not have a
tags file. Example: >

```
:psearch popen
```

This will show the "stdio.h" file in the preview window, with the function
prototype for popen():

```
FILE    *popen __P((const char *, const char *)); ~
```

You can specify the height of the preview window, when it is opened, with the
'previewheight' option.

==========

29.3 Moving through a program

Since a program is structured, Vim can recognize items in it. Specific
commands can be used to move around.
   C programs often contain constructs like this:

```
#ifdef USE_POPEN ~
    fd = popen("ls", "r") ~
#else ~
    fd = fopen("tmp", "w") ~
#endif ~
```

But then much longer, and possibly nested. Position the cursor on the
"#ifdef" and press %. Vim will jump to the "#else". Pressing % again takes
you to the "#endif". Another % takes you to the "#ifdef" again.
   When the construct is nested, Vim will find the matching items. This is a
good way to check if you didn't forget an "#endif".
   When you are somewhere inside a "#if" - "#endif", you can jump to the start
of it with: >

```
[#
```

If you are not after a "#if" or "#ifdef" Vim will beep. To jump forward to
the next "#else" or "#endif" use: >

```
]#
```

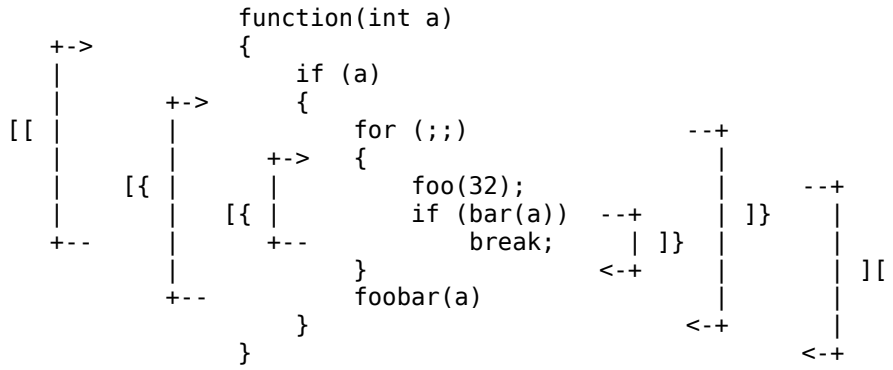
These two commands skip any "#if" - "#endif" blocks that they encounter.
Example:

```
#if defined(HAS_INC_H) ~
    a = a + inc(); ~
# ifdef USE_THEME ~
    a += 3; ~
# endif ~
    set_width(a); ~
```

With the cursor in the last line, "[#" moves to the first line. The "#ifdef"
- "#endif" block in the middle is skipped.

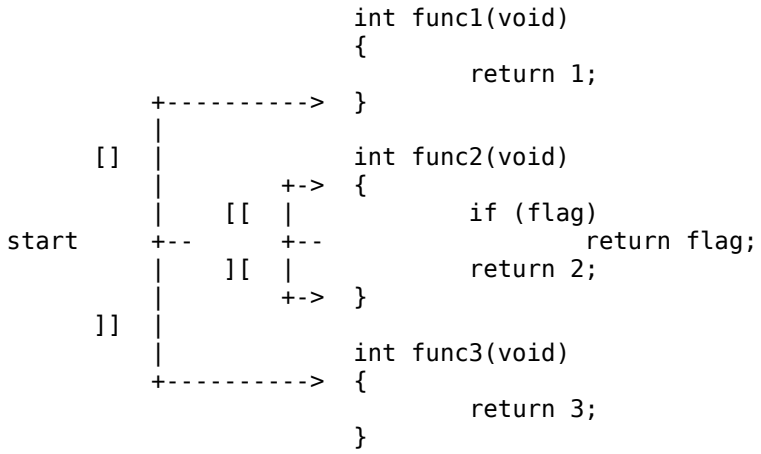
MOVING IN CODE BLOCKS

In C code blocks are enclosed in {}. These can get pretty long. To move to the start of the outer block use the "[[" command. Use "][" to find the end. This assumes that the "{" and "}" are in the first column.
The "[{" command moves to the start of the current block. It skips over pairs of {} at the same level. "]}" jumps to the end.
An overview:



When writing C++ or Java, the outer {} block is for the class. The next level of {} is for a method. When somewhere inside a class use "[m" to find the previous start of a method. "]m" finds the next start of a method.

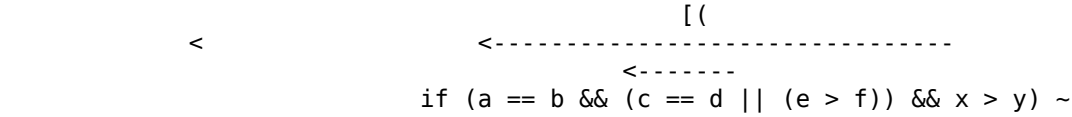
Additionally, "[]" moves backward to the end of a function and "]]" moves forward to the start of the next function. The end of a function is defined by a "}" in the first column.



Don't forget you can also use "%" to move between matching (), {} and []. That also works when they are many lines apart.

MOVING IN BRACES

The "[(" and "])" commands work similar to "[{" and "]}", except that they work on () pairs instead of {} pairs.
>



```
                              -------------->
                          ------------------------------> >
                                       ])
```

MOVING IN COMMENTS

To move back to the start of a comment use "[/". Move forward to the end of a comment with "]/". This only works for /* - */ comments.

```
          +->     +-> /*
          |    [/ |    * A comment about      --+
       [/ |       +--  * wonderful life.        | ]/
          |            */                     <-+
          |
          +--          foo = bar * 3;         --+
                                                | ]/
                       /* a short comment */  <-+
```

==========

29.4 Finding global identifiers

You are editing a C program and wonder if a variable is declared as "int" or "unsigned". A quick way to find this is with the "[I" command.
   Suppose the cursor is on the word "column". Type: >

```
        [I
```

Vim will list the matching lines it can find. Not only in the current file, but also in all included files (and files included in them, etc.). The result looks like this:

```
        structs.h ~
         1:   29     unsigned     column;    /* column number */ ~
```

The advantage over using tags or the preview window is that included files are searched. In most cases this results in the right declaration to be found. Also when the tags file is out of date. Also when you don't have tags for the included files.
   However, a few things must be right for "[I" to do its work. First of all, the 'include' option must specify how a file is included. The default value works for C and C++. For other languages you will have to change it.

LOCATING INCLUDED FILES

   Vim will find included files in the places specified with the 'path' option. If a directory is missing, some include files will not be found. You can discover this with this command: >

```
        :checkpath
```

It will list the include files that could not be found. Also files included by the files that could be found. An example of the output:

```
        --- Included files not found in path --- ~
        <io.h> ~
        vim.h --> ~
          <functions.h> ~
          <clib/exec_protos.h> ~
```

The "io.h" file is included by the current file and can't be found. "vim.h" can be found, thus ":checkpath" goes into this file and checks what it

includes. The "functions.h" and "clib/exec_protos.h" files, included by
"vim.h" are not found.

        Note:
        Vim is not a compiler. It does not recognize "#ifdef" statements.
        This means every "#include" statement is used, also when it comes
        after "#if NEVER".

To fix the files that could not be found, add a directory to the 'path'
option. A good place to find out about this is the Makefile. Look out for
lines that contain "-I" items, like "-I/usr/local/X11". To add this directory
use: >

        :set path+=/usr/local/X11

When there are many subdirectories, you can use the "*" wildcard. Example: >

        :set path+=/usr/*/include

This would find files in "/usr/local/include" as well as "/usr/X11/include".

When working on a project with a whole nested tree of included files, the "**"
items is useful. This will search down in all subdirectories. Example: >

        :set path+=/projects/invent/**/include

This will find files in the directories:

        /projects/invent/include ~
        /projects/invent/main/include ~
        /projects/invent/main/os/include ~
        etc.

There are even more possibilities. Check out the 'path' option for info.
   If you want to see which included files are actually found, use this
command: >

        :checkpath!

You will get a (very long) list of included files, the files they include, and
so on. To shorten the list a bit, Vim shows "(Already listed)" for files that
were found before and doesn't list the included files in there again.

JUMPING TO A MATCH

"[I" produces a list with only one line of text. When you want to have a
closer look at the first item, you can jump to that line with the command: >

        [<Tab>

You can also use "[CTRL-I", since CTRL-I is the same as pressing <Tab>.

The list that "[I" produces has a number at the start of each line. When you
want to jump to another item than the first one, type the number first: >

        3[<Tab>

Will jump to the third item in the list. Remember that you can use CTRL-O to
jump back to where you started from.

```
RELATED COMMANDS

        [i              only lists the first match
        ]I              only lists items below the cursor
        ]i              only lists the first item below the cursor


FINDING DEFINED IDENTIFIERS

The "[I" command finds any identifier.  To find only macros, defined with
"#define" use: >

        [D

Again, this searches in included files.  The 'define' option specifies what a
line looks like that defines the items for "[D".  You could change it to make
it work with other languages than C or C++.
   The commands related to "[D" are:

        [d              only lists the first match
        ]D              only lists items below the cursor
        ]d              only lists the first item below the cursor

==============================================================================
*29.5*  Finding local identifiers

The "[I" command searches included files.  To search in the current file only,
and jump to the first place where the word under the cursor is used: >

        gD

Hint: Goto Definition.  This command is very useful to find a variable or
function that was declared locally ("static", in C terms).  Example (cursor on
"counter"):

           +->   static int counter = 0;
           |
           |     int get_counter(void)
        gD |     {
           |         ++counter;
           +--       return counter;
                 }

To restrict the search even further, and look only in the current function,
use this command: >

        gd

This will go back to the start of the current function and find the first
occurrence of the word under the cursor.  Actually, it searches backwards to
an empty line above a "{" in the first column.  From there it searches forward
for the identifier.  Example (cursor on "idx"):

                int find_entry(char *name)
                {
           +->      int idx;
           |
        gd |        for (idx = 0; idx < table_len; ++idx)
           |            if (strcmp(table[idx].name, name) == 0)
           +--              return idx;
                }
```

==

Next chapter: |usr_30.txt| Editing programs

usr_30.txt   For Vim version 8.0. Last change: 2007 Nov 10

VIM USER MANUAL - by Bram Moolenaar

Editing programs

Vim has various commands that aid in writing computer programs. Compile a
program and directly jump to reported errors. Automatically set the indent
for many languages and format comments.

Next chapter: |usr_31.txt| Exploiting the GUI
Previous chapter: |usr_29.txt| Moving through programs
Table of contents: |usr_toc.txt|

==
30.1 Compiling

Vim has a set of so called "quickfix" commands. They enable you to compile a
program from within Vim and then go through the errors generated and fix them
(hopefully). You can then recompile and fix any new errors that are found
until finally your program compiles without any error.

The following command runs the program "make" (supplying it with any argument
you give) and captures the results: >

```
:make {arguments}
```

If errors were generated, they are captured and the editor positions you where
the first error occurred.
   Take a look at an example ":make" session. (Typical :make sessions generate
far more errors and fewer stupid ones.) After typing ":make" the screen looks
like this:

```
:!make | &tee /tmp/vim215953.err ~
gcc -g -Wall -o prog main.c sub.c ~
main.c: In function 'main': ~
main.c:6: too many arguments to function 'do_sub' ~
main.c: At top level: ~
main.c:10: parse error before '}' ~
make: *** [prog] Error 1 ~

2 returned ~
"main.c" 11L, 111C ~
(3 of 6): too many arguments to function 'do_sub' ~
Press ENTER or type command to continue ~
```

From this you can see that you have errors in the file "main.c". When you
press <Enter>, Vim displays the file "main.c", with the cursor positioned on
line 6, the first line with an error. You did not need to specify the file or

the line number, Vim knew where to go by looking in the error messages.

```
                +--------------------------------------------------+
                |int main()                                        |
                |{                                                 |
                |       int i=3;                                   |
      cursor -> |       do_sub("foo");                             |
                |       ++i;                                       |
                |       return (0);                                |
                |}                                                 |
                |}                                                 |
                | ~                                                |
                |(3 of 12): too many arguments to function 'do_sub' |
                +--------------------------------------------------+
```

The following command goes to where the next error occurs: >

```
        :cnext
```

Vim jumps to line 10, the last line in the file, where there is an extra '}'.
   When there is not enough room, Vim will shorten the error message. To see
the whole message use: >

```
        :cc
```

You can get an overview of all the error messages with the ":clist" command.
The output looks like this: >

```
        :clist
<       3 main.c: 6:too many arguments to function 'do_sub' ~
        5 main.c: 10:parse error before '}' ~
```

Only the lines where Vim recognized a file name and line number are listed
here. It assumes those are the interesting lines and the rest is just boring
messages. However, sometimes unrecognized lines do contain something you want
to see. Output from the linker, for example, about an undefined function.
To see all the messages add a "!" to the command: >

```
        :clist!
<       1 gcc -g -Wall -o prog main.c sub.c ~
        2 main.c: In function 'main': ~
        3 main.c:6: too many arguments to function 'do_sub' ~
        4 main.c: At top level: ~
        5 main.c:10: parse error before '}' ~
        6 make: *** [prog] Error 1 ~
```

Vim will highlight the current error. To go back to the previous error, use:
>

```
        :cprevious
```

Other commands to move around in the error list:

```
        :cfirst         to first error
        :clast          to last error
        :cc 3           to error nr 3
```

USING ANOTHER COMPILER

The name of the program to run when the ":make" command is executed is defined
by the 'makeprg' option. Usually this is set to "make", but Visual C++ users
should set this to "nmake" by executing the following command: >

```
        :set makeprg=nmake

You can also include arguments in this option.  Special characters need to
be escaped with a backslash.  Example: >

        :set makeprg=nmake\ -f\ project.mak

You can include special Vim keywords in the command specification.  The %
character expands to the name of the current file.  So if you execute the
command: >
        :set makeprg=make\ %:S

When you are editing main.c, then ":make" executes the following command: >

        make main.c

This is not too useful, so you will refine the command a little and use the :r
(root) modifier: >

        :set makeprg=make\ %:r:S.o

Now the command executed is as follows: >

        make main.o

More about these modifiers here: |filename-modifiers|.


OLD ERROR LISTS

Suppose you ":make" a program.  There is a warning message in one file and an
error message in another.  You fix the error and use ":make" again to check if
it was really fixed.  Now you want to look at the warning message.  It doesn't
show up in the last error list, since the file with the warning wasn't
compiled again.  You can go back to the previous error list with: >

        :colder

Then use ":clist" and ":cc {nr}" to jump to the place with the warning.
   To go forward to the next error list: >

        :cnewer

Vim remembers ten error lists.


SWITCHING COMPILERS

You have to tell Vim what format the error messages are that your compiler
produces.  This is done with the 'errorformat' option.  The syntax of this
option is quite complicated and it can be made to fit almost any compiler.
You can find the explanation here: |errorformat|.

You might be using various different compilers.  Setting the 'makeprg' option,
and especially the 'errorformat' each time is not easy.  Vim offers a simple
method for this.  For example, to switch to using the Microsoft Visual C++
compiler: >

        :compiler msvc

This will find the Vim script for the "msvc" compiler and set the appropriate
```

options.
   You can write your own compiler files. See |write-compiler-plugin|.

OUTPUT REDIRECTION

The ":make" command redirects the output of the executed program to an error file. How this works depends on various things, such as the 'shell'. If your ":make" command doesn't capture the output, check the 'makeef' and 'shellpipe' options. The 'shellquote' and 'shellxquote' options might also matter.

In case you can't get ":make" to redirect the file for you, an alternative is to compile the program in another window and redirect the output into a file. Then have Vim read this file with: >

```
:cfile {filename}
```

Jumping to errors will work like with the ":make" command.

==
30.2 Indenting C style text

A program is much easier to understand when the lines have been properly indented. Vim offers various ways to make this less work. For C or C style programs like Java or C++, set the 'cindent' option. Vim knows a lot about C programs and will try very hard to automatically set the indent for you. Set the 'shiftwidth' option to the amount of spaces you want for a deeper level. Four spaces will work fine. One ":set" command will do it: >

```
:set cindent shiftwidth=4
```

With this option enabled, when you type something such as "if (x)", the next line will automatically be indented an additional level.

```
                                if (flag)
Automatic indent   --->             do_the_work();
Automatic unindent <--          if (other_flag) {
Automatic indent   --->             do_file();
keep indent                         do_some_more();
Automatic unindent <--          }
```

When you type something in curly braces ({}), the text will be indented at the start and unindented at the end. The unindenting will happen after typing the '}', since Vim can't guess what you are going to type.

One side effect of automatic indentation is that it helps you catch errors in your code early. When you type a } to finish a function, only to find that the automatic indentation gives it more indent than what you expected, there is probably a } missing. Use the "%" command to find out which { matches the } you typed.
   A missing) and ; also cause extra indent. Thus if you get more white space than you would expect, check the preceding lines.

When you have code that is badly formatted, or you inserted and deleted lines, you need to re-indent the lines. The "=" operator does this. The simplest form is: >

```
==
```

This indents the current line. Like with all operators, there are three ways to use it. In Visual mode "=" indents the selected lines. A useful text

object is "a{". This selects the current {} block. Thus, to re-indent the
code block the cursor is in: >

```
	=a{
```

I you have really badly indented code, you can re-indent the whole file with:
>

```
	gg=G
```

However, don't do this in files that have been carefully indented manually.
The automatic indenting does a good job, but in some situations you might want
to overrule it.

SETTING INDENT STYLE

Different people have different styles of indentation. By default Vim does a
pretty good job of indenting in a way that 90% of programmers do. There are
different styles, however; so if you want to, you can customize the
indentation style with the 'cinoptions' option.
   By default 'cinoptions' is empty and Vim uses the default style. You can
add various items where you want something different. For example, to make
curly braces be placed like this:

```
	if (flag) ~
	  { ~
	    i = 8; ~
	    j = 0; ~
	  } ~
```

Use this command: >

```
	:set cinoptions+={2
```

There are many of these items. See |cinoptions-values|.

==
30.3 Automatic indenting

You don't want to switch on the 'cindent' option manually every time you edit
a C file. This is how you make it work automatically: >

```
	:filetype indent on
```

Actually, this does a lot more than switching on 'cindent' for C files. First
of all, it enables detecting the type of a file. That's the same as what is
used for syntax highlighting.
   When the filetype is known, Vim will search for an indent file for this
type of file. The Vim distribution includes a number of these for various
programming languages. This indent file will then prepare for automatic
indenting specifically for this file.

If you don't like the automatic indenting, you can switch it off again: >

```
	:filetype indent off
```

If you don't like the indenting for one specific type of file, this is how you
avoid it. Create a file with just this one line: >

```
	:let b:did_indent = 1
```

Now you need to write this in a file with a specific name:

```
        {directory}/indent/{filetype}.vim

The {filetype} is the name of the file type, such as "cpp" or "java".  You can
see the exact name that Vim detected with this command: >

        :set filetype

In this file the output is:

        filetype=help ~

Thus you would use "help" for {filetype}.
   For the {directory} part you need to use your runtime directory.  Look at
the output of this command: >

        set runtimepath

Now use the first item, the name before the first comma.  Thus if the output
looks like this:

        runtimepath=~/.vim,/usr/local/share/vim/vim60/runtime,~/.vim/after ~

You use "~/.vim" for {directory}.  Then the resulting file name is:

        ~/.vim/indent/help.vim ~

Instead of switching the indenting off, you could write your own indent file.
How to do that is explained here: |indent-expression|.

==============================================================================
*30.4*  Other indenting

The most simple form of automatic indenting is with the 'autoindent' option.
It uses the indent from the previous line.  A bit smarter is the 'smartindent'
option.  This is useful for languages where no indent file is available.
'smartindent' is not as smart as 'cindent', but smarter than 'autoindent'.
   With 'smartindent' set, an extra level of indentation is added for each {
and removed for each }.  An extra level of indentation will also be added for
any of the words in the 'cinwords' option.  Lines that begin with # are
treated specially: all indentation is removed.  This is done so that
preprocessor directives will all start in column 1.  The indentation is
restored for the next line.


CORRECTING INDENTS

When you are using 'autoindent' or 'smartindent' to get the indent of the
previous line, there will be many times when you need to add or remove one
'shiftwidth' worth of indent.  A quick way to do this is using the CTRL-D and
CTRL-T commands in Insert mode.
   For example, you are typing a shell script that is supposed to look like
this:

        if test -n a; then ~
           echo a ~
           echo "-------" ~
        fi ~

Start off by setting these options: >

        :set autoindent shiftwidth=3
```

You start by typing the first line, <Enter> and the start of the second line:

```
        if test -n a; then ~
        echo ~
```

Now you see that you need an extra indent. Type CTRL-T. The result:

```
        if test -n a; then ~
           echo ~
```

The CTRL-T command, in Insert mode, adds one 'shiftwidth' to the indent, no matter where in the line you are.
   You continue typing the second line, <Enter> and the third line. This time the indent is OK. Then <Enter> and the last line. Now you have this:

```
        if test -n a; then ~
           echo a ~
           echo "-------" ~
           fi ~
```

To remove the superfluous indent in the last line press CTRL-D. This deletes one 'shiftwidth' worth of indent, no matter where you are in the line.
   When you are in Normal mode, you can use the ">>" and "<<" commands to shift lines. ">" and "<" are operators, thus you have the usual three ways to specify the lines you want to indent. A useful combination is: >

```
        >i{
```

This adds one indent to the current block of lines, inside {}. The { and } lines themselves are left unmodified. ">a{" includes them. In this example the cursor is on "printf":

```
        original text           after ">i{"             after ">a{"

        if (flag)               if (flag)               if (flag) ~
        {                       {                           { ~
        printf("yes");              printf("yes");          printf("yes"); ~
        flag = 0;                   flag = 0;               flag = 0;  ~
        }                       }                           } ~
```

==========

30.5 Tabs and spaces

'tabstop' is set to eight by default. Although you can change it, you quickly run into trouble later. Other programs won't know what tabstop value you used. They probably use the default value of eight, and your text suddenly looks very different. Also, most printers use a fixed tabstop value of eight. Thus it's best to keep 'tabstop' alone. (If you edit a file which was written with a different tabstop setting, see |25.3| for how to fix that.)
   For indenting lines in a program, using a multiple of eight spaces makes you quickly run into the right border of the window. Using a single space doesn't provide enough visual difference. Many people prefer to use four spaces, a good compromise.
   Since a <Tab> is eight spaces and you want to use an indent of four spaces, you can't use a <Tab> character to make your indent. There are two ways to handle this:

1. Use a mix of <Tab> and space characters. Since a <Tab> takes the place of
    eight spaces, you have fewer characters in your file. Inserting a <Tab>
    is quicker than eight spaces. Backspacing works faster as well.

2. Use spaces only. This avoids the trouble with programs that use a
   different tabstop value.

Fortunately, Vim supports both methods quite well.

SPACES AND TABS

If you are using a combination of tabs and spaces, you just edit normally.
The Vim defaults do a fine job of handling things.
   You can make life a little easier by setting the 'softtabstop' option.
This option tells Vim to make the <Tab> key look and feel as if tabs were set
at the value of 'softtabstop', but actually use a combination of tabs and
spaces.
   After you execute the following command, every time you press the <Tab> key
the cursor moves to the next 4-column boundary: >

        :set softtabstop=4

When you start in the first column and press <Tab>, you get 4 spaces inserted
in your text. The second time, Vim takes out the 4 spaces and puts in a <Tab>
(thus taking you to column 8). Thus Vim uses as many <Tab>s as possible, and
then fills up with spaces.
   When backspacing it works the other way around. A <BS> will always delete
the amount specified with 'softtabstop'. Then <Tab>s are used as many as
possible and spaces to fill the gap.
   The following shows what happens pressing <Tab> a few times, and then using
<BS>. A "." stands for a space and "------->" for a <Tab>.

        type   result ~
        <Tab>   ....
        <Tab><Tab>   ------->
        <Tab><Tab><Tab>   ------->....
        <Tab><Tab><Tab><BS>   ------->
        <Tab><Tab><Tab><BS><BS> ....

An alternative is to use the 'smarttab' option. When it's set, Vim uses
'shiftwidth' for a <Tab> typed in the indent of a line, and a real <Tab> when
typed after the first non-blank character. However, <BS> doesn't work like
with 'softtabstop'.

JUST SPACES

If you want absolutely no tabs in your file, you can set the 'expandtab'
option: >

        :set expandtab

When this option is set, the <Tab> key inserts a series of spaces. Thus you
get the same amount of white space as if a <Tab> character was inserted, but
there isn't a real <Tab> character in your file.
   The backspace key will delete each space by itself. Thus after typing one
<Tab> you have to press the <BS> key up to eight times to undo it. If you are
in the indent, pressing CTRL-D will be a lot quicker.

CHANGING TABS IN SPACES (AND BACK)

Setting 'expandtab' does not affect any existing tabs. In other words, any
tabs in the document remain tabs. If you want to convert tabs to spaces, use
the ":retab" command. Use these commands: >

```
        :set expandtab
        :%retab
```

Now Vim will have changed all indents to use spaces instead of tabs. However, all tabs that come after a non-blank character are kept. If you want these to be converted as well, add a !: >

```
        :%retab!
```

This is a little bit dangerous, because it can also change tabs inside a string. To check if these exist, you could use this: >

```
        /"[^"\t]*\t[^"]*"
```

It's recommended not to use hard tabs inside a string. Replace them with "\t" to avoid trouble.

The other way around works just as well: >

```
        :set noexpandtab
        :%retab!
```

==========

30.6 Formatting comments

One of the great things about Vim is that it understands comments. You can ask Vim to format a comment and it will do the right thing.
Suppose, for example, that you have the following comment:

```
        /* ~
         * This is a test ~
         * of the text formatting. ~
         */ ~
```

You then ask Vim to format it by positioning the cursor at the start of the comment and type: >

```
        gq]/
```

"gq" is the operator to format text. "]/" is the motion that takes you to the end of a comment. The result is:

```
        /* ~
         * This is a test of the text formatting. ~
         */ ~
```

Notice that Vim properly handled the beginning of each line.
An alternative is to select the text that is to be formatted in Visual mode and type "gq".

To add a new line to the comment, position the cursor on the middle line and press "o". The result looks like this:

```
        /* ~
         * This is a test of the text formatting. ~
         * ~
         */ ~
```

Vim has automatically inserted a star and a space for you. Now you can type the comment text. When it gets longer than 'textwidth', Vim will break the line. Again, the star is inserted automatically:

```
        /* ~
         * This is a test of the text formatting. ~
         * Typing a lot of text here will make Vim ~
         * break ~
         */ ~
```

For this to work some flags must be present in 'formatoptions':

```
        r       insert the star when typing <Enter> in Insert mode
        o       insert the star when using "o" or "O" in Normal mode
        c       break comment text according to 'textwidth'
```

See |fo-table| for more flags.

DEFINING A COMMENT

The 'comments' option defines what a comment looks like. Vim distinguishes between a single-line comment and a comment that has a different start, end and middle part.
   Many single-line comments start with a specific character. In C++ // is used, in Makefiles #, in Vim scripts ". For example, to make Vim understand C++ comments: >

```
        :set comments=://
```

The colon separates the flags of an item from the text by which the comment is recognized. The general form of an item in 'comments' is:

```
        {flags}:{text}
```

The {flags} part can be empty, as in this case.
   Several of these items can be concatenated, separated by commas. This allows recognizing different types of comments at the same time. For example, let's edit an e-mail message. When replying, the text that others wrote is preceded with ">" and "!" characters. This command would work: >

```
        :set comments=n:>,n:!
```

There are two items, one for comments starting with ">" and one for comments that start with "!". Both use the flag "n". This means that these comments nest. Thus a line starting with ">" may have another comment after the ">". This allows formatting a message like this:

```
        > ! Did you see that site? ~
        > ! It looks really great. ~
        > I don't like it. The ~
        > colors are terrible. ~
        What is the URL of that ~
        site? ~
```

Try setting 'textwidth' to a different value, e.g., 80, and format the text by Visually selecting it and typing "gq". The result is:

```
        > ! Did you see that site? It looks really great. ~
        > I don't like it. The colors are terrible. ~
        What is the URL of that site? ~
```

You will notice that Vim did not move text from one type of comment to another. The "I" in the second line would have fit at the end of the first line, but since that line starts with "> !" and the second line with ">", Vim

knows that this is a different kind of comment.

A THREE PART COMMENT

A C comment starts with "/*", has "*" in the middle and "*/" at the end. The entry in 'comments' for this looks like this: >

```
:set comments=s1:/*,mb:*,ex:*/
```

The start is defined with "s1:/*". The "s" indicates the start of a three-piece comment. The colon separates the flags from the text by which the comment is recognized: "/*". There is one flag: "1". This tells Vim that the middle part has an offset of one space.
The middle part "mb:*" starts with "m", which indicates it is a middle part. The "b" flag means that a blank must follow the text. Otherwise Vim would consider text like "*pointer" also to be the middle of a comment.
The end part "ex:*/" has the "e" for identification. The "x" flag has a special meaning. It means that after Vim automatically inserted a star, typing / will remove the extra space.

For more details see |format-comments|.

==

Next chapter: |usr_31.txt| Exploiting the GUI

usr_31.txt For Vim version 8.0. Last change: 2007 May 08

VIM USER MANUAL - by Bram Moolenaar

Exploiting the GUI

Vim works well in a terminal, but the GUI has a few extra items. A file browser can be used for commands that use a file. A dialog to make a choice between alternatives. Use keyboard shortcuts to access menu items quickly.

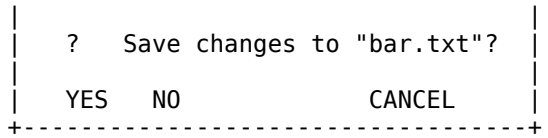
Next chapter: |usr_32.txt| The undo tree
Previous chapter: |usr_30.txt| Editing programs
Table of contents: |usr_toc.txt|

==
31.1 The file browser

When using the File/Open... menu you get a file browser. This makes it easier to find the file you want to edit. But what if you want to split a window to edit another file? There is no menu entry for this. You could first use Window/Split and then File/Open..., but that's more work.
Since you are typing most commands in Vim, opening the file browser with a typed command is possible as well. To make the split command use the file browser, prepend "browse": >

```
:browse split
```

```
Select a file and then the ":split" command will be executed with it.  If you
cancel the file dialog nothing happens, the window isn't split.
   You can also specify a file name argument.  This is used to tell the file
browser where to start.  Example: >

        :browse split /etc

The file browser will pop up, starting in the directory "/etc".

The ":browse" command can be prepended to just about any command that opens a
file.
   If no directory is specified, Vim will decide where to start the file
browser.  By default it uses the same directory as the last time.  Thus when
you used ":browse split" and selected a file in "/usr/local/share", the next
time you use a ":browse" it will start in "/usr/local/share" again.
   This can be changed with the 'browsedir' option.  It can have one of three
values:

        last            Use the last directory browsed (default)
        buffer          Use the same directory as the current buffer
        current         use the current directory

For example, when you are in the directory "/usr", editing the file
"/usr/local/share/readme", then the command: >

        :set browsedir=buffer
        :browse edit

Will start the browser in "/usr/local/share".  Alternatively: >

        :set browsedir=current
        :browse edit

Will start the browser in "/usr".

        Note:
        To avoid using the mouse, most file browsers offer using key presses
        to navigate.  Since this is different for every system, it is not
        explained here.  Vim uses a standard browser when possible, your
        system documentation should contain an explanation on the keyboard
        shortcuts somewhere.

When you are not using the GUI version, you could use the file explorer window
to select files like in a file browser.  However, this doesn't work for the
":browse" command.  See |netrw-browse|.

==============================================================================
*31.2*  Confirmation

Vim protects you from accidentally overwriting a file and other ways to lose
changes.  If you do something that might be a bad thing to do, Vim produces an
error message and suggests appending ! if you really want to do it.
   To avoid retyping the command with the !, you can make Vim give you a
dialog.  You can then press "OK" or "Cancel" to tell Vim what you want.
   For example, you are editing a file and made changes to it.  You start
editing another file with: >

        :confirm edit foo.txt

Vim will pop up a dialog that looks something like this:

        +----------------------------------+
```



Now make your choice. If you do want to save the changes, select "YES". If you want to lose the changes for ever: "NO". If you forgot what you were doing and want to check what really changed use "CANCEL". You will be back in the same file, with the changes still there.

Just like ":browse", the ":confirm" command can be prepended to most commands that edit another file. They can also be combined: >

```
:confirm browse edit
```

This will produce a dialog when the current buffer was changed. Then it will pop up a file browser to select the file to edit.

Note:
In the dialog you can use the keyboard to select the choice.
Typically the <Tab> key and the cursor keys change the choice.
Pressing <Enter> selects the choice. This depends on the system
though.

When you are not using the GUI, the ":confirm" command works as well. Instead of popping up a dialog, Vim will print the message at the bottom of the Vim window and ask you to press a key to make a choice. >

```
:confirm edit main.c
<       Save changes to "Untitled"? ~
        [Y]es, (N)o, (C)ancel:  ~
```

You can now press the single key for the choice. You don't have to press <Enter>, unlike other typing on the command line.

==========

31.3 Menu shortcuts

The keyboard is used for all Vim commands. The menus provide a simple way to select commands, without knowing what they are called. But you have to move your hand from the keyboard and grab the mouse.

Menus can often be selected with keys as well. This depends on your system, but most often it works this way. Use the <Alt> key in combination with the underlined letter of a menu. For example, <A-w> (<Alt> and w) pops up the Window menu.

In the Window menu, the "split" item has the p underlined. To select it, let go of the <Alt> key and press p.

After the first selection of a menu with the <Alt> key, you can use the cursor keys to move through the menus. <Right> selects a submenu and <left> closes it. <Esc> also closes a menu. <Enter> selects a menu item.

There is a conflict between using the <Alt> key to select menu items, and using <Alt> key combinations for mappings. The 'winaltkeys' option tells Vim what it should do with the <Alt> key.

The default value "menu" is the smart choice: If the key combination is a menu shortcut it can't be mapped. All other keys are available for mapping.

The value "no" doesn't use any <Alt> keys for the menus. Thus you must use the mouse for the menus, and all <Alt> keys can be mapped.

The value "yes" means that Vim will use any <Alt> keys for the menus. Some <Alt> key combinations may also do other things than selecting a menu.

```
==============================================================================
*31.4*  Vim window position and size

To see the current Vim window position on the screen use: >

        :winpos

This will only work in the GUI.  The output may look like this:

        Window position: X 272, Y 103 ~

The position is given in screen pixels.  Now you can use the numbers to move
Vim somewhere else.  For example, to move it to the left a hundred pixels: >

        :winpos 172 103
<
        Note:
        There may be a small offset between the reported position and where
        the window moves.  This is because of the border around the window.
        This is added by the window manager.

You can use this command in your startup script to position the window at a
specific position.

The size of the Vim window is computed in characters.  Thus this depends on
the size of the font being used.  You can see the current size with this
command: >

        :set lines columns

To change the size set the 'lines' and/or 'columns' options to a new value: >

        :set lines=50
        :set columns=80

Obtaining the size works in a terminal just like in the GUI.  Setting the size
is not possible in most terminals.

You can start the X-Windows version of gvim with an argument to specify the
size and position of the window: >

        gvim -geometry {width}x{height}+{x_offset}+{y_offset}

{width} and {height} are in characters, {x_offset} and {y_offset} are in
pixels.  Example: >

        gvim -geometry 80x25+100+300

==============================================================================
*31.5*  Various

You can use gvim to edit an e-mail message.  In your e-mail program you must
select gvim to be the editor for messages.  When you try that, you will
see that it doesn't work: The mail program thinks that editing is finished,
while gvim is still running!
   What happens is that gvim disconnects from the shell it was started in.
That is fine when you start gvim in a terminal, so that you can do other work
in that terminal.  But when you really want to wait for gvim to finish, you
must prevent it from disconnecting.  The "-f" argument does this: >

        gvim -f file.txt
```

The "-f" stands for foreground. Now Vim will block the shell it was started
in until you finish editing and exit.

DELAYED START OF THE GUI

On Unix it's possible to first start Vim in a terminal. That's useful if you
do various tasks in the same shell. If you are editing a file and decide you
want to use the GUI after all, you can start it with: >

	:gui

Vim will open the GUI window and no longer use the terminal. You can continue
using the terminal for something else. The "-f" argument is used here to run
the GUI in the foreground. You can also use ":gui -f".

THE GVIM STARTUP FILE

When gvim starts, it reads the gvimrc file. That's similar to the vimrc file
used when starting Vim. The gvimrc file can be used for settings and commands
that are only to be used when the GUI is going to be started. For example,
you can set the 'lines' option to set a different window size: >

	:set lines=55

You don't want to do this in a terminal, since its size is fixed (except for
an xterm that supports resizing).
   The gvimrc file is searched for in the same locations as the vimrc file.
Normally its name is "~/.gvimrc" for Unix and "$VIM/_gvimrc" for MS-Windows.
The $MYGVIMRC environment variable is set to it, thus you can use this command
to edit the file, if you have one: >

	:edit $MYGVIMRC
<
   If for some reason you don't want to use the normal gvimrc file, you can
specify another one with the "-U" argument: >

	gvim -U thisrc ...

That allows starting gvim for different kinds of editing. You could set
another font size, for example.
   To completely skip reading a gvimrc file: >

	gvim -U NONE ...

==

Next chapter: |usr_32.txt| The undo tree

Copyright: see |manual-copyright| vim:tw=78:ts=8:ft=help:norl:
usr_32.txt   For Vim version 8.0. Last change: 2010 Jul 20

		 VIM USER MANUAL - by Bram Moolenaar

			 The undo tree

Vim provides multi-level undo. If you undo a few changes and then make a new
change you create a branch in the undo tree. This text is about moving
through the branches.

     Next chapter: |usr_40.txt| Make new commands
 Previous chapter: |usr_31.txt| Exploiting the GUI
Table of contents: |usr_toc.txt|

==
32.1 Undo up to a file write

Sometimes you make several changes, and then discover you want to go back to
when you have last written the file. You can do that with this command: >

        :earlier 1f

The "f" stands for "file" here.

You can repeat this command to go further back in the past. Or use a count
different from 1 to go back faster.

If you go back too far, go forward again with: >

        :later 1f

Note that these commands really work in time sequence. This matters if you
made changes after undoing some changes. It's explained in the next section.

Also note that we are talking about text writes here. For writing the undo
information in a file see |undo-persistence|.

==
32.2 Numbering changes

In section |02.5| we only discussed one line of undo/redo. But it is also
possible to branch off. This happens when you undo a few changes and then
make a new change. The new changes become a branch in the undo tree.

Let's start with the text "one". The first change to make is to append
" too". And then move to the first 'o' and change it into 'w'. We then have
two changes, numbered 1 and 2, and three states of the text:

```
                one ~
                 |
              change 1
                 |
              one too ~
                 |
              change 2
                 |
              one two ~
```

If we now undo one change, back to "one too", and change "one" to "me" we
create a branch in the undo tree:

```
                one ~
                 |
              change 1
                 |
              one too ~
```

```
             /     \
          change 2  change 3
             |         |
          one two    me too ~
```

You can now use the |u| command to undo. If you do this twice you get to
"one". Use |CTRL-R| to redo, and you will go to "one too". One more |CTRL-R|
takes you to "me too". Thus undo and redo go up and down in the tree, using
the branch that was last used.

What matters here is the order in which the changes are made. Undo and redo
are not considered changes in this context. After each change you have a new
state of the text.

Note that only the changes are numbered, the text shown in the tree above has
no identifier. They are mostly referred to by the number of the change above
it. But sometimes by the number of one of the changes below it, especially
when moving up in the tree, so that you know which change was just undone.

==
32.3 Jumping around the tree

So how do you get to "one two" now? You can use this command: >

```
        :undo 2
```

The text is now "one two", you are below change 2. You can use the |:undo|
command to jump to below any change in the tree.

Now make another change: change "one" to "not":

```
                one ~
                 |
              change 1
                 |
              one too ~
              /     \
          change 2  change 3
             |         |
          one two    me too ~
             |
          change 4
             |
          not two ~
```

Now you change your mind and want to go back to "me too". Use the |g-|
command. This moves back in time. Thus it doesn't walk the tree upwards or
downwards, but goes to the change made before.

You can repeat |g-| and you will see the text change:

```
        me too ~
        one two ~
        one too ~
        one ~
```

Use |g+| to move forward in time:

```
        one ~
        one too ~
        one two ~
        me too ~
        not two ~
```

Using |:undo| is useful if you know what change you want to jump to. |g-| and
|g+| are useful if you don't know exactly what the change number is.

You can type a count before |g-| and |g+| to repeat them.

==
32.4 Time travelling

When you have been working on text for a while the tree grows to become big.
Then you may want to go to the text of some minutes ago.

To see what branches there are in the undo tree use this command: >

```
        :undolist
<       number changes  time ~
             3       2  16 seconds ago
             4       3  5 seconds ago
```

Here you can see the number of the leaves in each branch and when the change
was made. Assuming we are below change 4, at "not two", you can go back ten
seconds with this command: >

```
        :earlier 10s
```

Depending on how much time you took for the changes you end up at a certain
position in the tree. The |:earlier| command argument can be "m" for minutes,
"h" for hours and "d" for days. To go all the way back use a big number: >

```
        :earlier 100d
```

To travel forward in time again use the |:later| command: >

```
        :later 1m
```

The arguments are "s", "m" and "h", just like with |:earlier|.

If you want even more details, or want to manipulate the information, you can
use the |undotree()| function. To see what it returns: >

```
        :echo undotree()
```

==

Next chapter: |usr_40.txt| Make new commands